UNIVERSITY OF NEWCASTLE UPON TYNE
SCHOOL OF COMPUTING SCIENCE

CONTRACT REPRESENTATION FOR VALIDATION AND RUN TIME MONITORING

Ph.D. THESIS

By

Ellis Solaiman

NEWCASTLE UPON TYNE
MARCH 2004

NEWCASTLE UNIVERSITY LIBRARY

201 29900 9

Thesis L7581

Paginated blank pages are scanned as found in original thesis

No information is missing

Table of Contents

Acknowledgements....vii

Abstract....ix

1 ***Introduction***....1
- 1.1 Motivation....1
- 1.2 Research Background....2
- 1.3 Research Issues and Definitions....3
  - 1.3.1 Contracts, and x-contracts....3
  - 1.3.2 Contract validation....4
  - 1.3.3 Internal and External business processes....4
  - 1.3.4 Contract representation using finite state machines (FSMs)....5
  - 1.3.5 Run Time Requirements....6
- 1.4 Research tasks and objectives....6
- 1.5 Thesis Overview....7

2 ***Analysis of Related Work***....9
- 2.1 Open Distributed Programming Reference Model....9
- 2.2 CrossFlow....11
  - 2.2.1 CrossFlow Architecture....13
  - 2.2.2 Contract Creation....13
  - 2.2.3 Contract Enactment....14
- 2.3 The COYOTE Project....14
- 2.4 Work at Qeensland University....16
- 2.5 Work at University of St. Gallen....19
  - 2.5.1 Electronic Market Reference Model (EM-RM)....20
  - 2.5.2 The Business Media Framework....21
  - 2.5.3 Secure Contract Container SeCo....23
- 2.6 Electronic Commerce Development and Execution Environment (EDEE) 25
- 2.7 Law Governed Interaction....26
- 2.8 InterProcs....28
- 2.9 COSMOS....29
- 2.10 Trade intermediaries....31
- 2.11 Legal contracts as processes....31
- 2.12 Event-Trigger-Rules....32
- 2.13 Ponder Policy Specification Language....34
- 2.14 E-Commerce Frameworks....39
  - 2.14.1 ebXML....39
  - 2.14.2 BizTalk....39
  - 2.14.3 Web Services....39
  - 2.14.4 GRID....40
  - 2.14.5 eCo Framework....41
- 2.15 Discussion....41

3 ***Electronic Contracts as Finite State Machines***....45
- 3.1 Contracts and X-Contracts....45
  - 3.1.1 Rights and obligations....46

3.2 Finite State Machines 47
3.3 Representing Contracts as Finite State Machines 48
3.3.1 Mapping Contract Clauses into FSMs 49
3.3.2 Description of a simple contract using FSMs 50
3.4 Monitoring and enforcement of x-contracts 53
3.4.1 Invocation of rights and obligations 53
3.4.2 Description, monitoring and enforcement of an x-contract 55
3.4.3 Ready to fill in, sign and enact x-contracts 57
3.5 Summary 59

4 Validation of Electronic Contracts 61
4.1 A Verification Language - Promela 61
4.2 A Verification System - Spin 63
4.3 XSpin 64
4.4 The Spin Simulator 65
4.5 The Spin Validator 66
4.6 Correctness Requirements 68
4.6.1 Assertions and system invariants 68
4.6.2 Deadlocks 69
4.6.3 Progress cycles and livelocks 69
4.6.4 Temporal claims 69
4.6.5 Safety and liveness properties 70
4.6.6 Cost of correctness requirements 70
4.7 Basic Verification of x-contracts 71
4.7.1 Contract before removal of ambiguities 72
4.7.2 Contract after removal of ambiguities 81
4.8 Correctness requirements and Contracts, Discussion 85
4.9 Common correctness requirements 91
4.10 Summary 94

5 Validation of Electronic Contracts: Examples 95
5.1 Contract for the supply of electronic goods 95
5.1.1 The Contract 96
5.1.2 Split of rights and obligations 97
5.1.3 The finite state machines (The x-contract model) 99
5.1.4 The Verification model 101
5.1.5 X-contract verification 106
5.2 Example of a contract for renting cars 113
5.2.1 Car Rental Contract 113
5.2.2 Parties' rights and obligations 115
5.2.3 The finite state machines 117
5.2.4 The Promela Model: 119
5.2.5 X-contract verification 123
5.3 Playing a game over a network 129
5.3.1 Rules of the game 130
5.3.2 Players' rights and obligations 130
5.3.3 The finite state machine 132
5.3.4 Games' FSMs in Promela 133
5.3.5 Game model verification 141
5.4 Summary 144

6 Middleware Support for X-Contract Implementation 145
6.1 Overview of B2BObjects middleware 145
6.2 B2BObjects API 147
6.3 X-Contract Implementation with B2BObjects 149
6.4 Purchaser/Supplier Example 151
6.4.1 Implementation of Supplier/Purchaser Example 153
6.5 Summary 159

7 Summary and Future Work 161
7.1 Contract Modelling with Finite State Machines (Chapter 3) 161
7.2 Validation of electronic contracts (Chapter 4) 161
7.3 Modelling and Verifying the Correctness of Contracts; Examples (Chapter 5) 163
7.4 Middleware Support for X-Contract Implementation (Chapter 6) 164
7.5 For Future Work 164

References 167

List of Figures

Fig 2.1 RM-ODP Viewpoints [RM-ODP] 10
Fig 2.2 A meta model for the ODP enterprise viewpoint language 10
Fig 2.3 CrossFlow Contract structure in EER notation [KGV00] 11
Fig 2.4 The CrossFlow Architecture 13
Fig 2.5 Contract structure and content [GM00] 17
Fig 2.6 A UML model of a contract [GM00] 18
Fig 2.7 Electronic Market Reference Model [LR98] 21
Fig 2.8 Business Media Framework [RSK99] 21
Fig 2.9 Contracting framework for the contracting services. [RSK99] 22
Fig 2.10 Structure of the SeCo container [GSG00] 23
Fig 2.11 Contracting process [GSG00] 24
Fig 2.12 SeCo Container Architecture 25
Fig 2.13 Interaction of agents in LGI 27
Fig 2.14 COSMOS contract model [GBW98] 30
Fig 3.1 Abstraction of the main elements of a contract. 46
Fig 3.2 Mapping of events, conditions and operations of a contract into a FSM state 50
Fig 3.3 Example contract between a Purchaser and a_Supplier for the purchase of goods 51
Fig 3.4 FSM Representation of an ambiguous contract for the_purchase of goods . 52
Fig 3.5 Interaction of two FSMs by means of rights and obligations 53
Fig 3.6 FSM Representation of an unambiguous contract for the_purchase of goods 56
Fig 4.1 The Graphical user interface XSpin 64
Fig 4.2 The XSpin simulator interface 65
Fig 4.3 The Spin validator interface 67
Fig 4.4 Contract for the purchase of goods between a purchaser and a supplier 72
Fig 4.5 FSM Representation of an ambiguous contract for the_purchase of goods. 73
Fig 4.6 Message sequence chart 77
Fig 4.7 Simulation output 77
Fig 4.8 Verification Output 78
Fig 4.9 Suggested actions for detected error 79
Fig 4.10 Simulation output of erroneous path 79
Fig 4.11 FSM Representation of an unambiguous x-contract for the_purchase of goods 82
Fig 4.12 Verification output for the corrected verification model 84
Fig 4.13 Verification Output for detection of non-progress cycles 86
Fig 4.14 Verification output for detection of livelock (non-accept cycles) 88
Fig 4.15 Simulation output for path with livelock 88
Fig 4.16 The LTL Manager 91
Fig 5.3 Selection of general safety requirements for verification 107
Fig 5.4 Verification output for general safety properties 107
Fig 5.5 Verification of assertion claims 108
Fig 5.6 Message sequence chart, and Simulation_output of path with assertion violation 109
Fig 5.7 Verifier detects Livelock 111
Fig 5.8 Path through which Livelock was detected 112

Fig 5.9 Owner's Finite state machines 117
Fig 5.10 Renter's finite state machine 118
Fig 5.11 Safety error in the verification model 124
Fig 5.12 Verification output after checking general safety requirements 125
Fig 5.13 Simulation output through path where safety violation is detected 126
Fig 5.14 Verification options set to detect livelock 128
Fig 5.15 General safety error detected in Game model 141
Fig 5.16 Simulation output of path in which error is detected 141
Fig 5.17 Simulation of second problem in game model 142
Fig 5.18 Detection of unreachable code by the Spin validator. 144
Fig 6.1 B2Bobject Interactions 146
Fig 6.2 B2BObjects API 147
Fig 6.3 Collection of non-repudiable digital evidence with B2Bobjects 150
Fig 6.4 Contract clauses after removal of ambiguities 151
Fig 6.5 Corrected Purchaser and Supplier FMSs 152
Fig 6.6 Simple Example of a Contract Editor 157
Fig 6.7 Sample implementation of an x-contract 158

Acknowledgements

I am very grateful for all the support and encouragement that I have received during my research. I would like in particular to thank my Supervisor Professor Santosh Shrivastava for helping me decide which research topic to pursue and to proof-read this thesis and other papers. His experience and guidance have been invaluable. I also would like to give special thanks to Dr Carlos Molina-Jimenez for his huge contributions. His help and enthusiasm were a major motivating factor on the way to completing this work.

Many thanks also go to Richard Achmatowicz for his constant willingness to lend his support, which greatly accelerated my understanding of Spin and protocol validation.

Thanks go to Stuart Wheater for his help and suggestions, and Nick Cook for the development of the B2BObjects Middleware.

I would like to give thanks to various people within the department for their friendship Jonathan Burton especially when I started this work, Kamal Zamli for his humour, Jonathan Halliday, Doug Palmer, and Dimane Mpoeleng.

An essential contributing factor to this research was the support of my family, which I received in abundance from my parents Michael and Linda, my sister Amanda (Mandy), and my two brothers Samy and Barry.

During my time at Newcastle I have met some fantastic people who have simply been great friends; Victoria Tsismenaki (Vicky), Yaroslav Segal-Namir (Yarik), Daniel Sokolov , Martin Shaw, Mathew Dean, Angelique Anthian, Anna Maria Arango, James Renwick, Izara Khumium, Leonardo Bello, Joao de Silva, Franchesca Venezia, Mudassa Naqvi, Dimitiris, Craig Rose. Many thanks to you all.

This research has been funded by the UK Engineering and Physical Sciences Research Council (EPSRC).

Blank Page

Abstract

Organisations are increasingly using the Internet to offer their own services and to utilise the services of others. This naturally leads to resource sharing across organisational boundaries. Nevertheless, organisations will require their interactions with other organisations to be strictly controlled. In the paper-based world, business interactions, information exchange and sharing have been conducted under the control of contracts that the organisations sign. The world of electronic business needs to emulate electronic equivalents of the contract based business management practices.

This thesis examines how a 'conventional' contract can be converted into its electronic equivalent and how it can be used for controlling business interactions taking place through computer messages. To implement a contract electronically, a conventional text contract needs to be described in a mathematically precise notation so that the description can be subjected to rigorous analysis and freed from the ambiguities that the original human-oriented text is likely to contain. Furthermore, a suitable run time infrastructure is required for monitoring the executable version of the contract.

To address these issues, this thesis describes how standard conventional contracts can be converted into Finite State Machines (FSMs). It is illustrated how to map the rights and obligations extracted from the clauses of the contract into the states, transition and output functions, and input and output symbols of a FSM.

The thesis then goes on to develop a list of correctness properties that a typical executable business contract should satisfy. A contract model should be validated against *safety* properties, which specify situations that the contract must not get into (such as deadlocks, unreachable states ....etc), and *liveness* properties, which detail qualities that would be desirable for the contract to contain (responsiveness, accessibility ....etc). The FSM description can then be subjected to model checking. This is demonstrated with the aid of examples using the Promela language and the Spin validator.

Subsequently, the FSM representation can be used to ensure that the clauses stipulated in the contract are observed when the contract is executed. The requirements of a suitable run time infrastructure for monitoring contract compliance are discussed and a prototype middleware implementation is presented.

Chapter One
Introduction

1.1 Motivation

Over the past decade, increasing use of the Internet for commercial purposes has led what has become commonly known as *electronic commerce* (or *e-commerce*), to changing much of the traditional ways through which we conduct business. The key appealing qualities that are driving the Internet (and mainly the World Wide Web) to developing into the major business medium that it is increasingly becoming, and at such rapidity, are its simplicity of application, its global reach, and the speed at which it allows its users to interact. These factors are enabling business transactions to be performed at a higher efficiency rate than ever before, thus reducing the costs and efforts involved, and therefore improving organisations' business goals and aspirations.

The Internet, an application of computer and networking technologies, has itself become a technology on top of which applications are being constantly developed in order to enable and improve the ability of businesses to utilize the benefits which it promises.

As organisations increasingly use the Internet for their dealings, they will require their Internet business interactions with other organisations to be strictly monitored and controlled. The precondition of business interactions is a requirement of guarded trust between all business partners [GJS99]. In the paper-based world of commerce, to realize this vital precondition, business interactions, information exchange and sharing have been conducted under the control of contracts that organisations sign.

The world of electronic business needs to emulate electronic equivalents of the contract based business management practices. Internet applications should give different businesses entities, the possibility to make use of contracts electronically in a similar way that contracts are used conventionally.

There have been a number of attempts by different research groups at creating such applications, but tools and technologies for electronic management of contracts so far are not yet well developed.

1.2 Research Background

Initially, and until recent times, businesses that engaged in e-commerce conducted business interactions electronically solely over closed networks by means of Electronic Data Interchange (EDI) technology [S01]. Electronic Data Interchange defines a standard format for exchanging business data, which was developed by the Data Interchange Standards Association. It was first developed for US shipping and transportation industries in late 1970's, to reduce the burden of paperwork, a significant factor in cost during business transactions [EDI03].

In the early 1990's specialists identified EDI with electronic contracting, considering it as a term that solely refers to electronic transactions and contracts [AS03], and even as a shorthand acronym for electronic contracting [BP98].

An EDI message contains a string of data elements, each of which represents a singular fact, such as a price, product model number, and so forth, separated by delimiters. The entire string is called a data segment. One or more data segments framed by a header and trailer form a transaction set, which is the EDI unit of transmission (equivalent to a message). A transaction set often consists of what would usually be contained in a typical business document or form [SCIO03]. Traditional applications of EDI are purchase orders, bills, invoices, shipping orders and payments.

Today, two sets of standards governing the format of EDI are being used in the world. The first standard was developed by the UN; the WPFITP UN/EDIFACT syntax rules. The second was created by ASC X12 in the USA. Thus, while ASC X12 standards dominate in the USA, the UN/EDIFACT standards are more widely used elsewhere. The disparity between these standards is considered as a main obstacle for development of international EDI and has led to ongoing efforts (not yet successful) to harmonize the two sets of standards [S02].

With the development of the Internet, new concepts such as *closed and open* electronic contracting have acquired general acceptance. *Closed* electronic contracting can be defined as the use of EDI to expedite contracting among parties that already have a trading relationship established. *Open* electronic contracting allows the formation of contracts among parties with no prior trading relationships and is sometimes known as "arm's length" transactions [S02].

To facilitate contracting over the Internet, the international EDI community since 1995 has been developing Open-EDI, which is considered as a neutral framework for the future architecture of EDI on the Internet. It is proposed that Open-EDI will enable organizations to establish short-term relationships quickly and cost effectively. It will provide the opportunity to lower significantly the barriers to electronic data exchange by introducing standard business scenarios and the necessary services to support them [OEDI]. In principle,

once a business scenario is agreed upon, and implementations conform to the Open-EDI standards, there will be no need for prior agreement among trading partners, other than the decision to engage into the Open-EDI transaction in compliance with the business scenario. The field of application of Open-EDI will be the electronic processing of business transactions among autonomous multiple organizations within and across public, private, industrial, or geographic sectors. It will include business transactions that involve multiple data types such as numbers, characters, images and sound.

In addition to the EDI community, since the mid 1990's, electronic contracting has been the focus of many research groups and projects within business and academia, to name a few of these: Qeensland University [MB95], LGI [MU00], CrossFlow[CF00], COSMOS [CO99], and others. All of which have, and are contributing using different methods to enable business partners control their interactions by means of electronic contracts. Despite these concrete efforts, there are numerous unresolved issues that must be addressed if electronic contracting (e-contracting) is to be truly realised.

1.3 Research Issues and Definitions

1.3.1 Contracts, and x-contracts

We define a *conventional contract* as a document that stipulates that its signatories (two or more) agree to observe the clauses stipulated in the document. An electronic contract, or as we term it in this thesis; an *"executable contract (x-contract)"*, is the electronic version of a conventional contract and consists of one or more executable files complemented with zero or more ancillary files (text, graphics, images, etc.), that can be enacted to enforce what the English text contract stipulates [MSSW03*]. Each entry in a contract is called a *term* or a *clause*. The clauses of a contract stipulate how the signing parties are expected to behave. In other words, they list the *rights* and *obligations* of each signing party. A *right* is an action that a signing entity can exercise if it wishes to. For example, a contract might stipulate that Alice, as a manager of enterprise E1, has the right to send an offer to sell to Bob, the manager of enterprise E2. Because this is a right, it is up to Alice to send or not to send the offer to Bob; Bob need not be disappointed if he does not receive the offer. Similarly, an *obligation* is defined as a duty that an entity is expected to perform. A failure to perform such a duty means a breach of the contract. For example, a contract might stipulate that upon receiving an offer to sell from Alice, Bob has the obligation to reply to her with an *OfferAccepted* or *OfferRejected* message.

Hard-copy paper based contracts have been used for a long time we know therefore how to write (for example in English), interpret and execute a conventional contract. Unfortunately, contracts in the electronic world are not yet well understood. In particular, converting a conventional contract into an executable contract is not a trivial process.

This thesis examines how relevant parts of a *conventional* contract can be converted into its electronic equivalent and how it can be used for controlling business interactions taking place between computers connected over the Internet.

In our work, contracts are conceptually located between the interacting parties and are meant to drive the execution of inter-enterprise business processes.

1.3.2 Contract validation

We identify a crucial difference between conventional contracts, and x-contracts. A conventional contract is human oriented. Thus, it is likely to contain ambiguities in the text that are detected and interpreted by humans when the contract is performed; whereas an x-contract is computer oriented; consequently, it tolerates no inconsistencies. Therefore, to implement a contract electronically, a conventional text contract needs to be described in a mathematically precise notation so that the description can be subjected to precise analysis and freed from the ambiguities that the original human-oriented text is likely to contain.

1.3.3 Internal and External business processes

An organization's business processes can be divided into two broad categories: the business processes that are *internal* to the organization, and the *external* contract management processes that involve interactions with trading partners (a business process whether internal or external, is defined as a set of organized activities aiming at reaching a common business goal [R98].).

There has been and still is a great deal of research into the automation of an organisation's internal processes. A variety of computer systems for automating the task of scheduling and executing application have been developed. These systems are known as workflow management systems and the applications are called workflows. Research into the automation of an organisation's external processes however, has been fairly recent in comparison.

This thesis addresses the issue of facilitating the control and monitoring of an organisation's external processes, through the specification, validation, and monitoring of electronic contracts.

In our business model [MSSW03] enterprises that engage in contractual relationships are autonomous and wish to remain autonomous after signing a contract. Thus a signing

enterprise has its own resources and local policies. In our view each contracting enterprise is a black box where private business processes represented as finite state machines, workflows or similar automaton, run. A private business process interacts with its external environment through the contract from time to time to influence the course of the shared business process. Thus, a contract is a mechanism that is conceptually located in the middle of the interacting enterprises to intercept all the contractual operations that the parties try to perform. Intercepted operations are accepted or rejected in accordance with the contract clauses and role players' authentication.

From this perspective, we can identify two fairly independent sources of contract inconsistencies:

- Internal enterprise policies conflicting with contractual clauses.
- Inconsistencies in the clauses of the contract.

It is our view that these two issues should be treated separately rather than encumbering a contract model with excessive notation (details, concepts and information) that might be extremely difficult to validate. Such a separation is not considered in the work of most research groups. In this Thesis we address only the second issue, that is, we are concerned only with the cooperative behaviour of business enterprises and not their internal structure.

1.3.4 Contract representation using finite state machines (FSMs)

To address the issues that we have raised in the previous sections of this chapter, this thesis proposes that contracts be converted into finite state machines (FSMs). We have found that a finite state machine is a simple yet expressive model for describing, validating and implementing x-contracts. We will describe how standard conventional contracts can be translated into Finite State Machines (FSMs), and illustrate how to map the rights and obligations extracted from the clauses of a contract into the states, transition and output functions, and input and output symbols of a FSM. We develop and suggest a list of correctness properties that a typical electronic business contract should satisfy, and demonstrate a verification process through which a FSM representation of a contract can be validated for correctness with respect to the correctness properties that we suggest. Finally, we demonstrate how a validated FSM can be used to ensure that the clauses in a contract are observed when the contract is executed.

1.3.5 Run Time Requirements

The mere fact that organisations require the use of contracts to regulate their interactions with each other, leads us to the assumption that they do not completely trust each other. So an important requirement from the middleware that will facilitate these interactions is that it should enable regulated interactions (as encoded in the x-contract) between two or more mutually suspicious but autonomous organizations. It is clearly not possible to prevent organisations from misbehaving and attempting to cheat on their agreed contractual relationships. The best that can be achieved is to ensure that all contractual interactions between such organisations are funnelled through (a centralised or distributed) contract management system and that either (a) all other non-contractual interactions are disallowed, or (b) the contract management system is at least capable of monitoring and signalling the signatories of the contract as to when the contract is being violated, or ideally (c) both a, and b.

The *safety* properties of the middleware must ensure that local policies of an organization are not compromised despite failures and/or misbehavior by other parties; whilst the *liveness* properties should ensure that if all the parties are correct (not misbehaving), then agreed interactions would take place despite a bounded number of temporary network and computer related failures. Also because we are dealing with contracts, for the purposes of proof and legality the middleware must have means for collecting non-repudiable evidence of the actions of parties that interact with each other.

In this thesis, the requirements of a suitable run time infrastructure for monitoring contract compliance are discussed and a prototype middleware implementation is presented.

1.4 Research tasks and objectives

We take as an input an existing conventional text based business contract. The reason we say "existing" is because we do not investigate within this thesis how to negotiate contracts over the Internet, we assume that a contract has already been negotiated, and that it already exists. Using it we show how a contract can be segmented into the basic rights and obligations that form it. We go on to demonstrate a method by which these extracted rights and obligations can be used as the parameters of mathematical models. The mathematical models we use for this process are finite state machines.

Once the contract has been converted into finite state machines, we need to make sure that the FSM model of the contract is not ambiguous. This means that we need to make sure that the FSM representation of the contract has been designed accurately, and that it performs the operations that the contract is expected to perform, and does not perform (or detects and

signals) operations that the contract is not expected to perform. This interpretation of ambiguity however is too vague.

Our next task therefore will be to define a set of standard *correctness requirements* that can be used by the x-contract model designer as a guide for removing any ambiguity within the x-contract model. The tool that we use and propose for this process is the model checker Spin [SP03].

We next show how a correct (free from ambiguity) FSM contract model, can be coded into the x-contract. The programming language we use for this purpose is JAVA. The JAVA FSM contract (or the x-contract) therefore will be used for the purpose of monitoring the interactions between the signatories to the contract. We discuss further the requirements (discussed in Section 1.3.5) of the middleware service on top of which the x-contract is to be implemented, and propose B2BObjects (developed at Newcastle University) [CSW02] as an appropriate middleware service that serves these requirements.

We finally demonstrate how the x-contract and the middleware can be used together to facilitate the electronic contracting process. We demonstrate these operations with the aid of a number of examples.

1.5 Thesis Overview

In Chapter 2, we present and discuss the research relevant to ours that has been conducted in the area of electronic contracts.

In Chapter 3, we define finite state machines, and take a look at some of their applications. We demonstrate how they can be used to represent conventional contracts, and how they can be used for monitoring and enforcing the clauses within a contract.

The verification of the correctness of protocols and how it relates to the validation of x-contracts is discussed in Chapter 4. We present and describe the model checker Spin that can be used for removing the ambiguities within contracts. Also in Chapter 4 we suggest a set of correctness requirements against which the correction of a contract can be validated, and introduce the concepts of *safety* and *liveness* requirements. In Chapter 5, we present a number of different examples of contracts to demonstrate the contract conversion and verification process.

The requirements from a suitable middleware service for the implementation of x-contracts are discussed in Chapter 6. In this chapter we propose the use of *B2BObjects* as a middleware service and use it for implementing x-contracts. Finally in Chapter 7, we close with some conclusions.

Blank Page

Chapter Two

Analysis of Related Work

In this Chapter, we present work that is relevant to our research work conducted by various academic, and industry based research groups in the area of electronic contracting. We analyse different works while focusing mainly on efforts in contract representation, contract validation, contract monitoring, and electronic contract implementation.

We begin our discussion with the Reference Model of Open Distributed Processing (RM-ODP). It is a joint effort between the ISO (International Standards Organisation), and the ITU (International Telecommunication Union). This work is born from recognising a requirement of a coordinating framework for the standardisation of Open Distributed Processing [RM-ODP].

The CrossFlow project is presented next. Central to its architecture are contracts, which are used to connect the work flow management systems (WFMS) of different cooperating organisations within a virtual environment, resulting in a Cross Organisational WFMS. Next we Continue with industry based research; we take a look at IBM's COYOTE project. After this we analyse the work done at a number of universities including Milosovic et al at Queensland University, and Naftaly Minsky et al at Rutgers University.

We examine different approaches while keeping in mind the contract implementation requirements, and the goals that we outlined in Section 1.4.

2.1 Open Distributed Programming Reference Model

ODP describes systems that support heterogeneous distributed processing both within and between organisations through the use of a common interaction model. The Reference Model prescribes a framework using five "viewpoints" (abstractions); enterprise, information, computational, engineering, and technology.

A set of concepts, structures, and rules is given for each of the viewpoints, providing a language for specifying ODP systems in that viewpoint [RM-ODP]. It is hoped that using each of the five viewpoint languages, a large and complex specification of an ODP system can be separated into manageable pieces, each focused on the issues relevant to different members of a development team. Fig 2.1, shows how the RM-ODP viewpoints can be related to the software engineering process [RM-ODP].

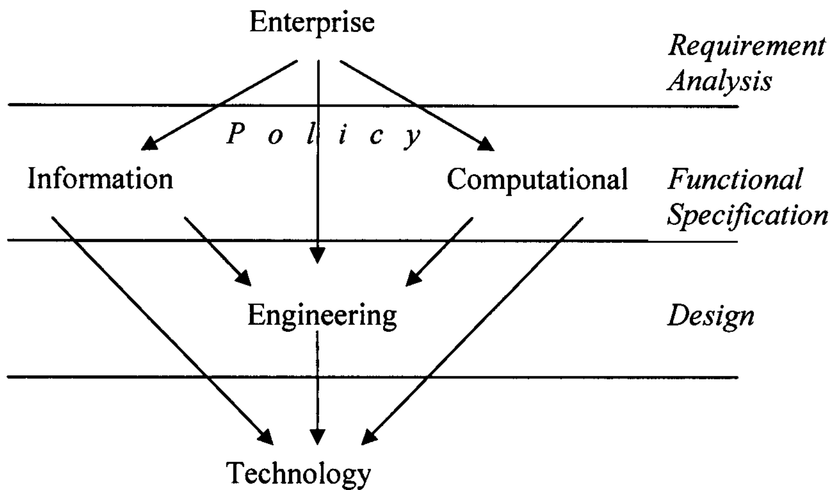


Fig 2.1. RM-ODP Viewpoints [RM-ODP]

The enterprise viewpoint language incorporates concepts such as policies and roles within a community. Policies are defined in terms of; *Objects* (bank managers, customers, money, bank accounts, etc), *communities* (grouping of *objects* formed to meet an objective. The objective is expressed as a contract, which specifies how the objective can be met), and roles of objects expressed in terms of policies (permission, obligation, and prohibition).

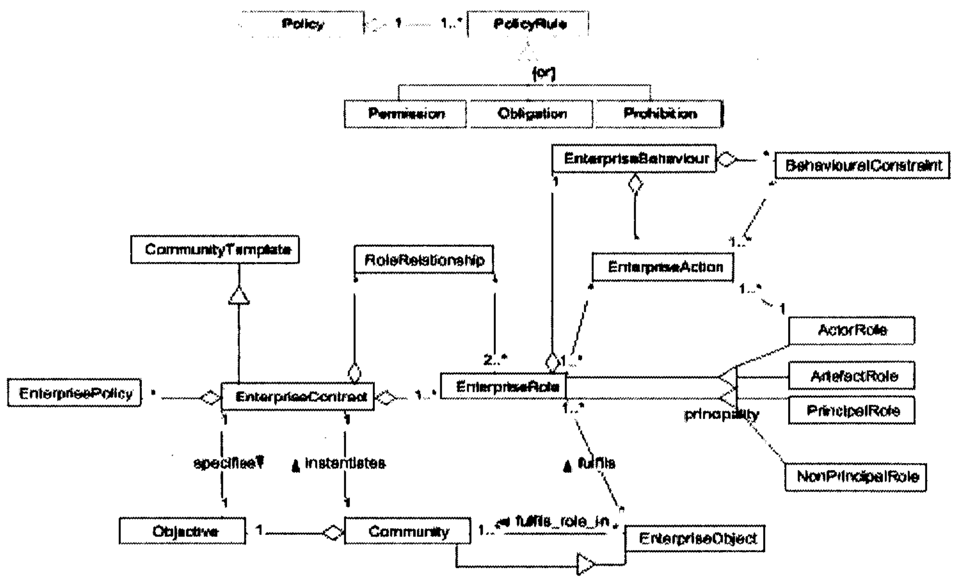


Fig.2.2. A meta model for the ODP enterprise viewpoint language [SD00]

Policies constrain the behaviour of enterprise objects that fulfil actor roles in communities and are designed to meet the objective of the community. Policy specifications define what behaviour is allowed or not allowed and often contain prescriptions of what to do when a rule is violated.

The ODP enterprise language is really a set of abstract concepts rather than a language that can be used to specify enterprise policies and roles. Recently, there have been a number of attempts to define precise languages that implement the abstract concepts of the enterprise language. These approaches concentrate on using UML to graphically depict the static structure of the enterprise viewpoint language as exemplified by [SD00] (see figure 2.2), as well as languages to express policies based on those UML models.

2.2 CrossFlow

The CrossFlow project [CF00] aims at developing concepts and information technology for advanced workflow support in virtual organizations that are dynamically formed by contract based service trading [KGV00].

Contracts are used in CrossFlow for flexible service outsourcing, in which a service provider organization performs a service on behalf of a service consumer organization. Contracts are the basis for finding suitable partners, establishing business relationships, connecting work flow management systems (WFMS) of different kinds, controlling outsourced workflows, and sharing abstractions of workflow specifications between partners. Contracts in CrossFlow, define all data, process elements and enactment conditions relevant to the co-operation through the outsourced workflow process on an abstract level.

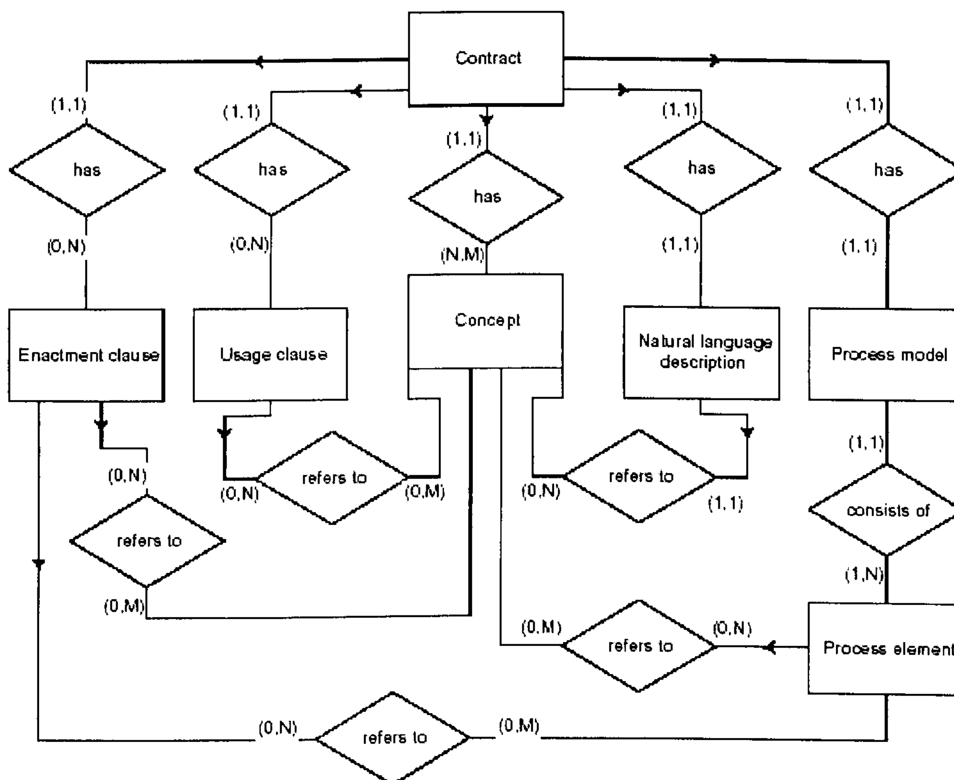


Fig 2.3. CrossFlow Contract structure in EER notation [KGV00]

The establishment of virtual enterprises with the help of contracts is discussed in [HLGG00]. Contracts are established automatically by CrossFlow contract manager modules without human interaction. Negotiation of contracts is not required in the context of the project and thus not covered by the approach. The approach is based on standard form contracts that describe standardized services in the context of specific markets.

The data structure of the CrossFlow contract model in EER (Extended Entity Relationship) notation can be seen in figure 2.3. In the figure rectangles describe elements, and diamonds describe the type of relationship between the elements. Details on reading EER notation can be found in [EER]. The model consists of five main elements:

1. The *Concept Model* establishes the terminology of the contract. The concepts of the contract are defined as a list of parameters that can have complex structures. The parameters are defined with their name, type and description. The concept model consists of three parts. *General parameters* describe attributes that are applicable to contracts in general. This part standardizes contracts by ensuring that parameters used in any service always have the same name and structure, like CONSUMER, PROVIDER, and SERVICENAME. Having this part makes it easier to search for a contract on generally accepted terms. *Service specific parameters* apply to specific service types. Parameters like DELIVERY ADDRESS, PACKAGEWEIGHT, etc. are only applicable to transport services. The consumer should specify the values of these parameters in the contract, so the provider process can read them from the contract and start the workflow instance. *Process variables* are dynamic parameters used for exchanging information during the service execution.

2. The *Process Model* describes the internal structure of the workflow process implementing the service. The process is composed of process elements, e.g. the individual activities and transitions. The process needs to be specified in a way that allows the provider to map it to its actual process and allows the consumer to understand the sequence of events and make decisions based on this knowledge.

3. The *Enactment Model.* The enactment clauses in the contract define additional enactment requirements on top of basic workflow processing defined in the workflow definition. Enactment clauses can be related to enactment performance monitoring, cross-organizational process control, advanced transaction management, automatic remuneration, etc.

4. The *Usage Model* defines how contracts are used for service outsourcing. These definitions are related to the concept of Partially Filled Contracts as explained below.

5. The *Natural Language Description* is a piece of text that is not meant for electronic interpretation, but for human reading. This text can be used to describe the service in an understandable way and to refer to the legal context of the transaction.

To enable the use of standard contracts, the CrossFlow approach has defined the concept of Partially Filled Contracts (PFCs). PFCs are contract templates of which the service specific fields are partially filled by a service consumer. On the basis of the PFC, a business agreement is reached between service provider and consumer for the enactment of multiple services. Each service is specified by completing the PFC to a complete filled contract. A life cycle model has been defined relating various kinds of templates, PFCs and actual contracts.

2.2.1 CrossFlow Architecture

The CrossFlow architecture [CF00] supports both contract creation and contract (service) enactment. The architecture is based on commercial workflow management system technology, shielded from the CrossFlow technology by an interface layer. In the project, IBM's MQSeries workflow product is used [MQ].

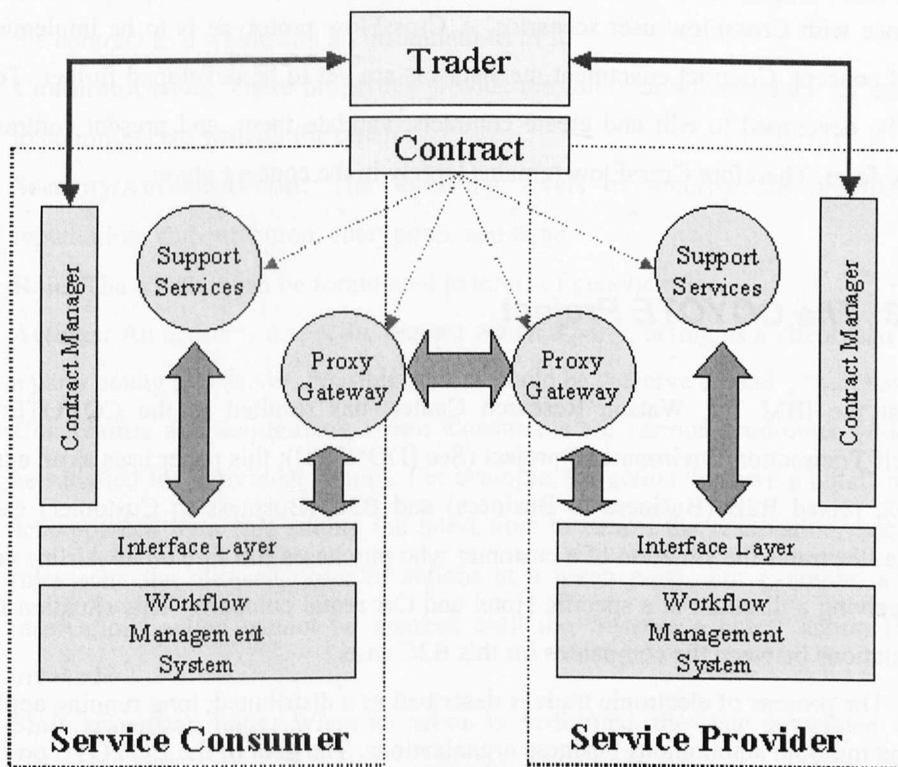


Fig.2.4. The CrossFlow Architecture

2.2.2 Contract Creation

When a service provider wants to advertise a service it can perform on another organization's behalf, it uses its contract manager to send a contract template to a trader. When a service consumer wants to outsource the enactment of a service, it uses a contract template to search

for service providers via a trader. When a match between consumer's requirements and provider's offer is found, an electronic contract can be made by filling in the template.

2.2.3 Contract Enactment

Based on specifications in the contract, a dynamic contract and service enactment architecture is set up. The symmetrical architecture contains proxy gateways that control all communication and support services for advanced cooperation functionality. After contract completion, the dynamically created modules can be disposed of.

An XML based contract language has been developed for the CrossFlow project. The language can be used to define contracts that are suitable for supporting fully automated outsourcing transactions. Using the language, it is hoped that all aspects of the interaction can be defined in a structured manner, enabling the co-operation infrastructure to set up and manage the outsourcing environment. A prototype of the cooperation infrastructure is under development. In the future, the model and language will be refined and extended, based on the experience with CrossFlow user scenarios. A CrossFlow prototype is to be implemented as proof of concept. Contract enactment mechanisms are yet to be developed further. Tools are still to be developed to edit and create contracts, validate them, and present contracts in a readable form. Therefore CrossFlow remains largely in the concept phase.

2.3 The COYOTE Project

Work at the IBM T.J. Watson Research Center, has resulted in the COYOTE (Cover YOurself Transaction Environment) project (See [DDNS98]); this paper uses as an example a complex, mixed B2B (Business to Business) and B2C (Business to Customer) case. The example illustrates the situation of a customer who purchases full fare on an Airline company thus receiving a discount at a specific Hotel and Car rental company. This situation involves close relations between the companies for this B2C case.

The process of electronic trade is described as a distributed, long running application, spanning multiple autonomous business organizations. The goal of the COYOTE project is to provide an application development and execution environment for electronic business applications. The approach makes a clear separation of internal and external business processes within an organization. The rules of external interaction and externally visible states are defined as a service contract. Service contracts act both as a guideline for interaction across businesses and also as an enforcement mechanism for guaranteeing proper interaction. It is a high-level description of the interaction between two or more contracting parties. The contract contains two kinds of information. The first kind is a machine-readable description of

the computer-to-computer interactions between the parties that supports the overall application. It concerns those aspects of the application that each party must agree with and which are enforceable by the COYOTE system. The second is the usual human-readable legal language that is part of any business-to-business contract and includes those aspects of the agreement, which must be enforced by the person to person contact.

The project is concerned with the machine-readable section. The contract is written in XML. After reaching an agreement by all parties, the XML contract can then be turned into a code. This code is called a service-contract object (SCO) and resides at each of the parties. It provides interfaces to the application programs of the parties. Each party communicates via the SCO of the other party. The service contract defines the properties of each party that must be made visible to the other parties to the contract. These properties include:

Identification: The identification section assigns a name to the contract and provides the names of each of the parties to the contract.

Overall properties: The overall properties are attributes of the contract that apply to the contract as a whole and all instantiations of it.

Communication: These properties provide the information necessary for each party to communicate with all the others.

Security/Authentication: The following levels of security are provided: non-repudiation, authentication, encryption, and none.

Role: The contract can be formulated in terms of generic roles.

Actions: An action is a specific request which a party, acting as a client, can issue to a party acting as a server. Possible actions could be "reserve a hotel", "purchase", etc.

Constraints and sequencing rules: Constraints are various conditions, which must be satisfied for individual actions. For example, the action "reserve a hotel" might be accompanied by a rule stating the latest time to cancel the reservation. Sequencing rules state the allowed order of actions at a given party. For example, a "cancel reservation" action cannot be invoked until the "reserve a hotel" action has been invoked.

State transition logic: When an action is performed, the state associated with the action (and hence the state of the contract) changes. The contract defines additional changes of variables and parameters, which take place following the completion or failure of an action.

Compensation rules: These properties state any conditions relating to the cancellation of previously invoked actions.

Error handling: These properties contain error conditions and methods to be called when they occur.

Legal aspects: These properties contain conditions, which are typically defined in a legal contract such as handling of disputes and other exceptional conditions.

These properties provide the structure of the service contract. This structure is comparable with the structure of the SeCo Container (Section 2.5). For example, the legal terms in the SeCo Container are identical to the legal aspects; the agents to the identity, etc.

A specific aspect of this project is support for error handling. The approach in this project is process oriented and the contracts must support the different processes that result from the activities of the companies. Thus the additional properties that are needed are state transition logic, actions and error handling.

Due to the diversity in the business processes, the many possible actions and responses from the parties in a business transaction, this paper considers the use of Petri Net like definitions of the business processes as inappropriate. This view contradicts [LS98] and [RL98], where Petri Nets are used for process modelling. COYOTE has not addressed the issue of validation of contracts.

2.4 Work at Qeensland University

The idea of monitoring and enforcement of policies specifically for electronic contracts has been discussed by Milosovic et al [GM00] [MM01] [MB95]. In [MB95], "A possible sequence of contract operations" is proposed. The sequence includes the "Establishment Phase" where the parties negotiate the terms of the contract and sign it, and the "Performance Phase", where the contract is monitored and enforced.

Contract monitoring, is defined as the process of observing the activities of a company and tracking these activities not to violate the contract. Monitoring can be performed by the parties or by a third party acting on behalf of one or all the parties. This process can be performed continuously during the contract execution or can occur from time to time. In case that one of the parties breaks the contract conditions, contract enforcement can take place.

In [MAO96], the focus is mainly on the security requirements for open distributed systems for contracts. Special attention is paid to the competence element. This is essentially the issue of determining if a given person has the authority to establish a contract. The proposal is to base competence on a notion of roles, which reflect the structure of a company, e.g. presidents, managers, and administrators. As is common practice, a person can proceed with a request if the permission is obtained from some collections of the superiors such as 3 managers or 2 state managers, or simply the permission of the president. Digital signatures

and roles are used to implement competence that can be verified by people within a company and by those negotiating with the company.

The paper [GM00] aims at the specification and implementation of business contracts needed for Business-to-Business (B2B) electronic commerce. In this approach, valid business contract must contain four elements; agreement, consideration, capacity, legal purpose. These elements result in clauses that cover items like; parties, definition and interpretation of terms, jurisdiction, etc. (see figure 2.5). In the appendix of [GM00] an example of such a contract is listed. We actually borrow this contract and modify it to adapt it to our particular needs as one of our examples in Chapter 5.

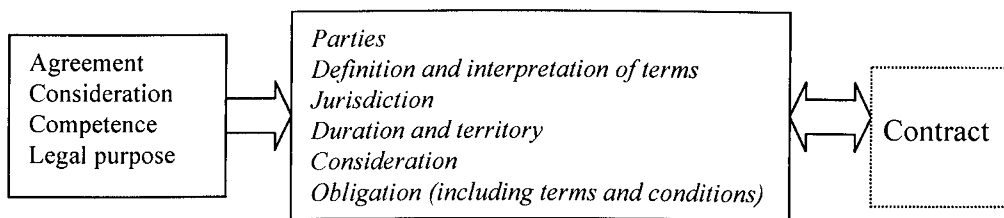


Fig.2.5. Contract structure and content [GM00]

In this work, a contract can be provided by one of the parties (e.g. the seller), a third party or a commercial organization that will provides general–purpose contracts for different business scenarios.

Also in [GM00], the authors pay particular attention to the benefits of using standard contracts, a concept that is also important in our work. A standard contract template can be provided by one of the parties (e.g. the seller), a third party or a commercial organization that will be providing general–purpose contracts for different business scenarios.

A number of basic roles are needed to support typical operations associated with contract establishment. The Contract Repository (CR) is needed to store standard form contracts and standard contract clauses. The Notary is used to store signed instances of standard form contracts, which can be used later as evidence of agreement in contract monitoring and enforcement activities. The Contract Validator (CV) performs the contract validity checking procedures relating to the legality of the contract. The Contract Monitor (CM) is used for monitoring the contract. The CM has three major roles: to monitor the party activities, to record and measure actions and performance, to deal with non-performance parties. In case of improper behaviour the CM informs the Contract Enforcer (CE) component. The Contract Enforcer (CE), upon being signalled by the CM, performs enforcing actions such as sending a message to various parties informing them of the violation and possibly preventing further access to the system by non-conforming parties. The Contract Negotiator

(CN) is an optional role that can be used to mediate the negotiation of contracts in the contractual phase.

The Contract Monitor has the central component CMM (Contract Monitor Manager), which receives policy statements of the form:

<policy> :: = <variable_declaration>
***when** <condition>*
<action>
***must [not] occur where** <condition>*
***otherwise** <trigger_action>;*
*<action> :: = **action(**<action_name>, <actor>,*
*<audience>,<time>,<body>**)***
*<trigger_action> ::= **trigger_action(**<action_name>,*
*<audience>,<body>**)***

Upon receiving a policy statement, it is analysed by the CMM, and the CE is signalled if a violation is detected.

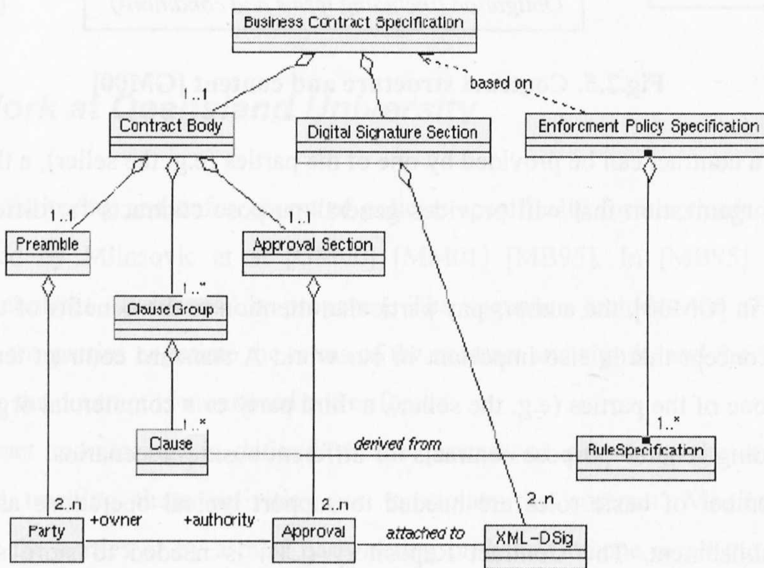


Fig.2.6. A UML model of a contract [GM00]

Based on this analysis the authors define a contract model. The contract model contains the following elements: a preamble that outlines the parties involved in the contract and the nature of the consideration; a clause element, which is a list of contract clauses, clustered in logical groups; an approval section that enumerates who from each party approved the contract; a digital signature section, with digital signatures from the appropriate parties listed in the approval section; a section containing a list of policy specifications stating contract enforcement rules according to the agreed contract clauses.

Contractual terms and conditions are modelled as policies. This is influenced by the Event-Condition-Action (ECA) paradigm from active databases, and the ODP language. Policies are embedded in the XML structure.

Validation of the correctness of contracts has been identified as crucial by Milocevic et al in [GM00], and [MD02], where a contract is informally defined as a set of policy statements that specify constraints in terms of permissions, prohibitions and obligations for roles involved in the contract. A role (precisely, a role player) is an entity (for example a human being, machine, program, etc.) that can perform an action. Formally, each policy statement is specified in deontic logic constraints [MM01]. Thus each deontic constraint precisely defines the permissions, prohibitions, obligations, actions, and temporal and non-temporal conditions that a role needs to fulfil to satisfy an expected behaviour.

For example, a constraint can formally specify that, "Bob is obliged to deliver a box of chocolates to Alice's desk every weekday except on Wednesdays for three years, between 9 and 9:15 am, commencing on the 1st of Jan 2004". The expressiveness of deontic notation allows the contract designer to verify temporal and deontic inconsistencies in the contract. The authors of this approach argue that it is possible to build verification software to visually show that, Bob's obligations do not overlap or conflict. Such verification mechanisms would easily detect a conflicting situation where Bob has to deliver a box of chocolates to Alice's desk and to Claire's who works miles away from Alice's desk. Similarly, the verifier would detect that Bob is not obliged and prohibited to deliver chocolates to Alice during the same period of time.

There are similarities between this research and ours in the focus on the use of standard contracts, the validation of the correctness of contracts prior to implementation, and in the monitoring of the x-contract at execution time. However whereas we focus only on the validation of the contract specified business to business interactions, Milocevic et al also include in their validation process the checking of the consistency of contract specified interactions with the internal processes of the signing entities. We consider this to be too ambitious. Also it is not very clear in this work if the collection of non-repudiable evidence is possible without involving trusted third parties.

2.5 Work at University of St. Gallen

A number of papers on electronic contracting have been written by two main research groups at the University of St Gallen. The main contributions of these papers are the Electronic Market Reference Model, the Business Media Framework, and the Secure Contract Container (SeCo).

2.5.1 Electronic Market Reference Model (EM-RM)

EM-RM is a model developed over a series of papers, which discuss a number of issues relevant to electronic contracting. In [LR97] non-repudiation within electronic contracting is the main topic. It addresses the issue of trust between parties by introducing *trust centers* - institutional instruments to support confidentiality and non-repudiation.

Trust Centers are independent third parties, in which a high extent of confidentiality is put. According to [FHK95] the term Trust Center aggregates trusted Third Parties (TTP) and Personal Trust Centers (PTC). *"A Trusted Third Party is an impartial organization delivering business confidence through commercial and technical security features, to an electronic transaction. It supplies technically and legally reliable means of carrying out, facilitating, producing independent evidence about and/or arbitrating on an electronic transaction. As services are provided and underwritten by technical, legal financial and/or structural means."* [LR97].

The idea of *trust centres* is gaining popularity [KBCS00], and is considered by many researchers and developers as one of the most probable solutions for the legal issue problems and the lack of trust between parties.

[LRP97] and [LR97] describe a *Model for Permanent IT-Support*. By *Permanent IT-support* it is meant continuous and time-independent information technology support throughout all phases of an electronic commerce transaction. The suggested model consists of a business layer, a services layer, and a technical layer.

In [LS98], [RA98] and [LR98], the model is developed, and the three layers are modified, and they become 4 so called "views". (See figure 2.7). To summarize, [LR98] proposes an Electronic Market Reference Model (EM-RM), which consists of two dimensions. The horizontal dimension contains the three phases of an electronic market transaction (Information phase, Agreement phase, and Settlement phase), whereas the vertical dimension is built of four views. The four views can be grouped into two main blocks of which the upper two views (Business and Transaction View) focus on o*rganizational aspects,* whilst the lower two views (Services and Infrastructure View) depict *technological aspects.*

Within the agreement phase, an *electronic contracting tool*, which includes functions for the purpose of negotiation of traditional trade via new electronic means, is suggested. The contracting tool in figure 2.7 is a framework for other individual contract negotiation supporting tools. Goal and content of the settlement phase are mainly the booking, payment, and delivery of ordered goods and services. In the agreement phase; negotiated, and in the settlement phase; concluded contracts, provide the basis and specification for these goals. The paper does not mention how these goals are to be arrived at from the contract, or whether the interactions between the parties are subsequently monitored to ensure the achievement of

these goals during the settlement phase. Also no attention is paid to the technologies that need to be involved in the construction of an electronic market.

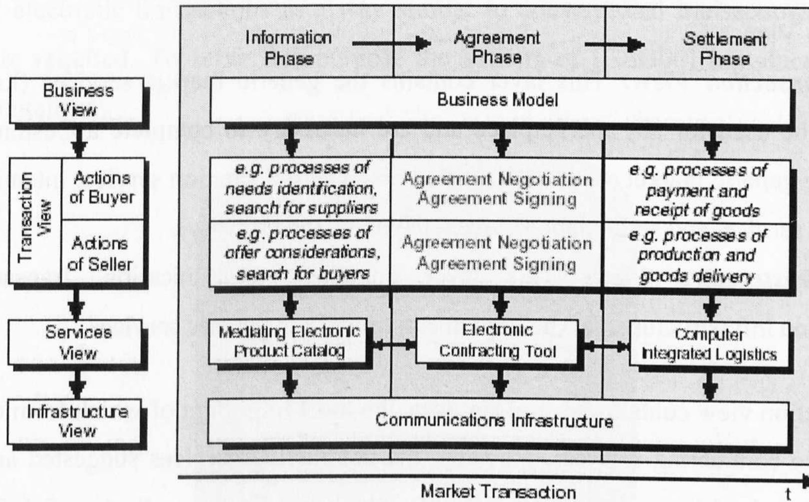


Fig.2.7. Electronic Market Reference Model [LR98]

2.5.2 The Business Media Framework

[RSK99] proposes a solution for the management of business transactions, considering contracts as the key information object of all legally relevant actions in a business transaction. A contracting framework is presented.

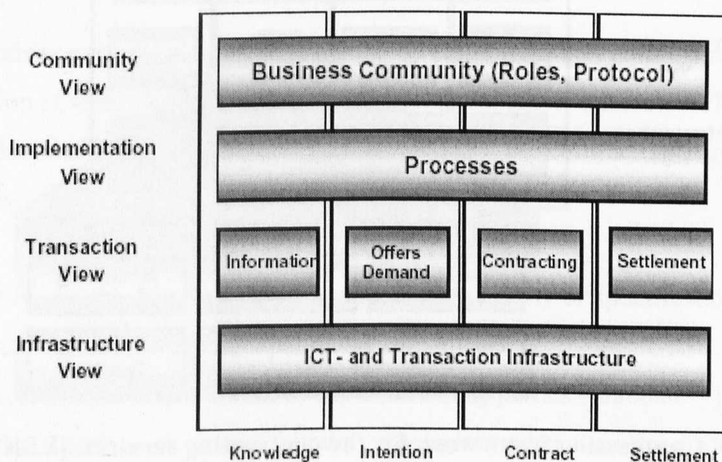


Fig.2.8. Business Media Framework [RSK99]

Four layers are identified in the BMF (Figure 2.8):

Community View: The interested business community is described and structured on this layer.

***Implementation View*:** On this layer the roles, protocols and processes that have been identified in the Community View are based on the underlying generic services of the Transaction View.

***Transaction View*:** This layer contains the generic market services (i.e. services, which can be used for any marketplace and are necessary to complete a Customer Buying Cycle). The generic market services identified are: The information service, intention service, contracting service, and settlement services (payment and delivery).

Infrastructure View: This layer contains communication, transaction, and transportation infrastructure, for the implementation of the generic services.

The transaction view contains several services, the most important of which from our point of view are the contracting services. However the contracting services suggested are based on the work done by Milocevic et al in [GM00], which we discuss in Section 2.4. The authors have used the defined BMF with the roles defined in [GM00] in order to build a Contracting Framework for the contracting services (see figure 2.9). This framework is a combination of the two models.

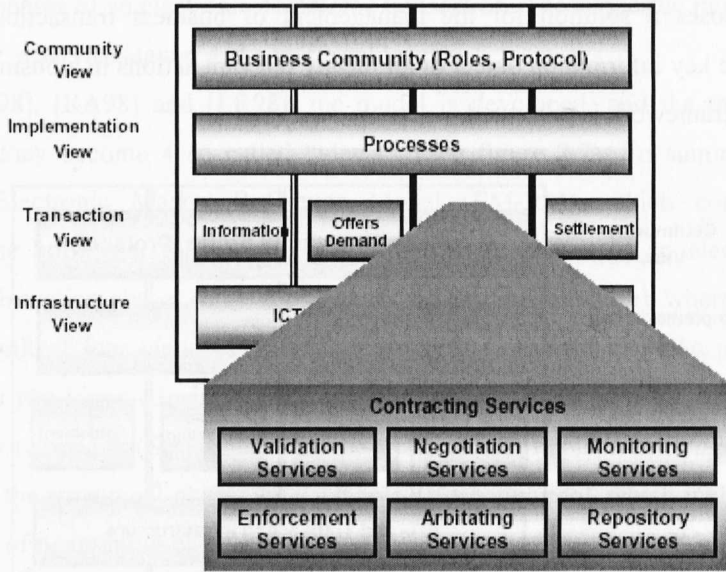


Fig.2.9. Contracting framework for the contracting services. [RSK99]

[GSG00] investigates the technological aspect as well as the legal aspect of Electronic Contracts. It focuses on contracts that involve two parties. The two roles in a contract are:

An *offeror* - the person who makes an offer.

An *offeree* - the person who receives an offer.

2.5.3 Secure Contract Container SeCo

To support electronic transactions in a way similar to conventional transactions, electronic contracts are required. To serve this purpose the authors of [GSG00] introduce a "Secure Contract Container".

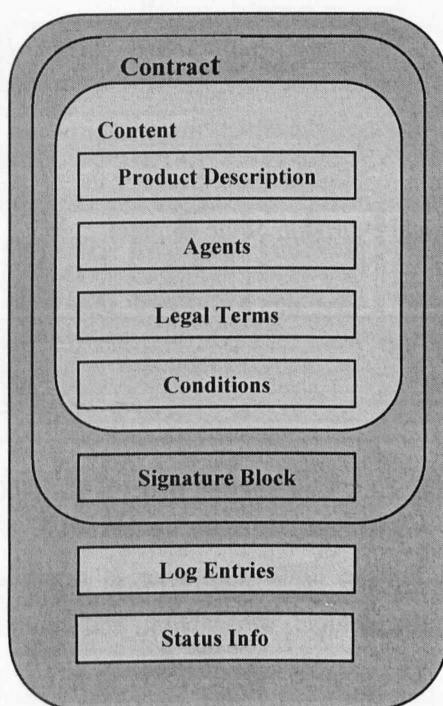


Fig.2.10. Structure of the SeCo container [GSG00]

A SeCo container comprises two parts: a contract section and an administrative section. The contract section is separated into a content section and a signature block. The *content section* contains all data that is relevant for the contract and that the contracting parties have to agree on. It includes:

- The product or service descriptions with agreed upon quality or specifications of all products and services the customer intends to purchase.
- The identification and address data of the contracting parties (mandatory), as well as other involved market agents such as an arbitrator, a recipient other than the customer, or a notary (optional). This sub-section is referred to as "Agents" (see figure 2.10).
- The legal terms of the contract as well as the arbitration code.
- The delivery and payment conditions together with the communication protocols applied in the integration of payment and logistics services (i.e. SET).

The contract content section in the SeCo Container serves the same purpose as the Concept Model in the CrossFlow contract model [KGV00].

The *signature block* holds the digital signatures signing the content section. Furthermore, the signature block contains the corresponding X509 certificates that hold the public keys of the signers (X509 is a standard for digital certificates). The *log section* logs the events that occur during the contracting process, as well as any relevant information that arises during the fulfilment of the contract. The *status section* holds information about the current state of the SeCo Container. It can be used as a quick reference for queries for the status of a contract. A container can hold more than one contract section, resulting from the process of negotiation, but at any given time, there is only one valid contract section. The most recent contract section represents the current state of the contracting process. This allows tracking of the historical evolution of the contract.

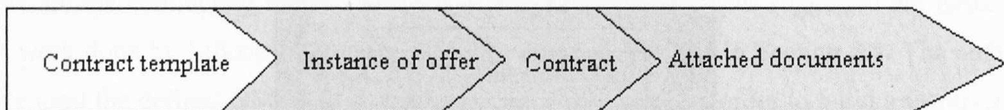


Fig.2.11. Contracting process [GSG00]

The software architecture for the SeCo Container is described in [GSS00]. The SeCo Container architecture is built of logic, information, and communication layers (see figure 2.12)

On the **Logic Layer**, the logic of the business transaction is designed, managed, and performed. The logic layer can manage the monitoring of the contracting process through checking critical dates and values, and through performing actions like reminding of the outstanding signing of the contract or the non-performance of the delivery. The logic layer has a secure access to the information structured on the information layer.

The **Information Layer** provides data storage and contains the contract information. The data of the information layer contains a structured and an unstructured part. In the structured part all the information that is subject to further processing in the contracting or settlement phase is stored. The structured part is divided into four blocks:

(1) Who block- The involved parties are described.

(2) What block- Product or service object of the contract is specified.

(3) How block- The settlement conditions of the transactions i.e. the enactment clauses.

(4) Legal block- The legal circumstances, under which the parties came to a mutual agreement.

In the unstructured part of the information layer, documents that are collected throughout a market transaction could be added. In order to have a document history it is proposed to generate a new document for each step of the contracting process. This new document inherits certain attributes either from the container settings or from already existing documents.

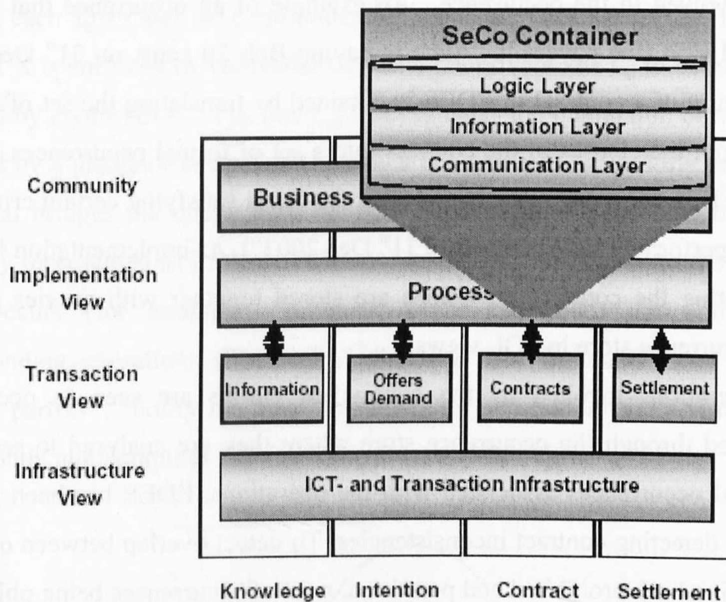


Fig. 2.12. SeCo Container Architecture

The **Communication Layer** includes all protocols necessary for the communication with the generic market services and the contracting parties.

To summarise, The SeCo container is able to collect and contain all the information that is necessary to enforce a contract in front of an arbitrating court (Information Layer). It contains the logic rules about the obligations that have to be fulfilled and about the security of its own state (Logic Layer). The authors do not however discuss whether there is a verification process in which an electronic contract is checked for correctness before implementation. Also there aren't any examples or details on the implementation of SeCo beyond the Architecture descriptions above.

2.6 Electronic Commerce Development and Execution Environment (EDEE)

Another research work of relevance to ours is the EDEE system. EDEE provides a framework for representing, storing and enforcing business contracts [AEB01].

In EDEE a contract is informally conceived as a set of provisions. In legal parlance, a provision is an arrangement in a legal document, thus in EDEE a provision specifies an obligation, prohibition, privilege or power (a privilege or power is equivalent to a right in our

work). An example of a provision is "Alice is obliged to pay Bob 20 cents before 1st Jan 2004". Central to EDEE is the concept of occurrence. An occurrence is a time-delimited relationship between entities. It can be regarded as a participant-occurrence-role triple that contain the name of the participants of the occurrence, the name of the occurrence and the name of the roles involved in the occurrence. An example of an occurrence that involves Alice (the payer) and Bob (the payee) is "Alice is paying Bob 20 cents on 31st Dec 2003." The formal specification of a contract in EDEE is obtained by translating the set of informal provisions derived from the clauses of the contract into a set of formal occurrences. Another basic concept in EDEE is query. A query is a request for items satisfying certain criteria (for example, "Payments performed by Alice before 31st Dec 2003"). At implementation level, the occurrences representing the contract provisions are stored together with queries and new occurrences in an occurrence store in SQL views.

Business operations invoked by the contractual parties are seen as occurrences intercepted and passed through the occurrence store where they are analysed to see if they satisfy the contractual occurrences associated with the operations. EDEE has been provided with some means for detecting contract inconsistencies. To detect overlap between queries (a set of occurrences being both prohibited and permitted, a set of occurrences being obliged and prohibited, etc.) the authors of EDEE rely on a locally implemented coverage-checking algorithms.

2.7 Law Governed Interaction

Electronic contracts have been studied by Naftaly Minsky, and his research group in a number of papers under the concept of Law Governed Interaction (LGI) [XMNU00] [UM00] [MMU01] [SXM01] [MU01].

The LGI mechanism is a message exchange software layer that allows a group of distributed agents to interact over a communication medium (see figure 2.13), honouring a set of previously agreed upon rules. An agent is an entity, for example, a computer program, with means for sending and receiving messages. As the term *agent* suggests, agents act on behalf of their enterprises. In the LGI paradigm, a business to business interaction involves a set of private laws and one interaction law: the private laws are internal to each enterprise and regulate the activities of the agents while operating as representatives of their enterprises whereas the interaction law is public to the members of the group and regulates the interactions between the enterprises. It is worth mentioning that the interaction law is actually the business contract that the agents are expected to honour when they interact with each other on behalf of their enterprises.

A law can be regarded as a set of rules. An example of a rule contained in a private law would be "Agent E_i can place purchase orders without the approval of the manager only for purchases not exceeding 5000 pounds."

Laws are enforced by controllers which are trusted entities conceptually placed between each agent and the communication medium. Thus the private law L_p to be honoured by agent X is enforced by controller C_x while the private law L_q to be honoured by agent Y is enforced by controller C_y. The law L_{xy} that regulates the interaction between agent X and Y is enforced by a mediator controller C_{xy} which is conceived as working on behalf of a mediator agent that bridges the interactions between agents X and Y. Every controller stores its law (formally represented as Prolog-like terms) and the current control state of its agent. When an event occurs (for example, "purchase order received") the controller performs the corresponding operations stipulated in the laws (for instance, "send acknowledgement to business partner", "notify the local manager", etc.) to honour the private law, the interaction law, or both, and computes the new control state [MU01].

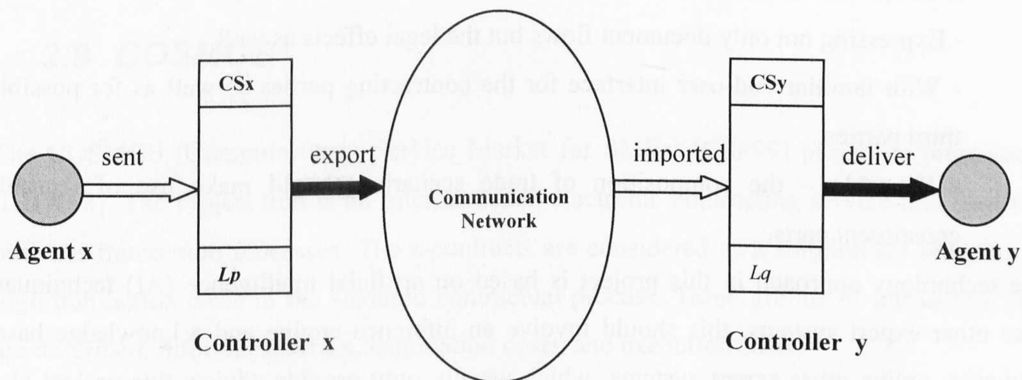


Fig.2.13. Interaction of agents in LGI

The LGI approach is similar to ours in that it suggests a separation of business to business laws from internal-to-enterprises ones. Likewise, the job of the mediator controller closely resembles the job of the FSMs of our approach. To the best of our knowledge, the LGI group has no reported results about validation of the laws or about how the controllers collect non-repudiable evidence of the operations performed by their agents.

Also what is not clear, is the implementation of the mediator controller Cxy that regulates the contract law Lxy. We will assume that the mediator agent that Cxy works for is a trusted third party that enforces Lxy.

2.8 InterProcs

Ronald M. Lee at the Erasmus University in Rotterdam in [L98] presents a design and pilot implementation of a system, supporting electronic contracting, called InterProcs. A key deliverable of this project is a model expert system for producing trade scenarios customized to a particular situation, yet making use of stored knowledge and experience on their design and legal controls. This is a generalization of the Open-EDI approach used in [LS98], where only standard trade scenarios are used. The project aims to provide an artificially intelligent framework for constructing trade scenarios. For this reason, the authors aim not only to understand the sequencing of document flows, but to understand why these documents are sent, and what the purpose of these documents is, i.e. their legal effects. The formal representation of the trade scenarios should be:

- Procedurally representable.
- Computable thus allowing fully automated computer-to-computer transactions.
- Customizable – parties should be able to customize the generic trade scenarios for their specific needs.
- Expressing not only document flows but the legal effects as well.
- With familiar end-user interface for the contracting parties as well as for possible third parties.
- Reusable – the composition of trade scenarios should make use of reusable constituent parts.

The technology approach in this project is based on artificial intelligence (AI) techniques. Like other expert systems, this should involve an inference engine and a knowledge base. However, unlike other expert systems, which usually only provide advice, this project also involves a transaction system, which is able to execute the trade scenarios automatically.

A key objective for the design of trade scenarios is the inclusion of appropriate documentary controls, e.g. protecting against fraud, accident or misinterpretation, and providing appropriate evidence of the contract status, should the contract come into dispute and go to court. These controls may be either detective, recognizing when something has gone wrong, or preventative, in avoiding the error in the first place. Two additional open challenges are listed:

revisability - while a given contract is being 'executed', it should be capable of revision (due to possible constraints previously set in the contract clauses).

evolvablility - the knowledge base of trade scenarios should be able to evolve, based on learning and experience from past modeling.

The contracting process in this paper is divided into three main phases: shopping, negotiation and performance. Though the terminology is different the three phases can be easily mapped

to information, agreement, settlement i.e. the phases discussed in the research of St. Galen University. A basic issue for this project is how electronic trade scenarios should be represented from the modeller's perspective, and from a computation perspective. In this paper Document Petri Nets (DPN's) are considered (see [LS98]) to be the most appropriate representation for capturing the temporal/dynamic aspects of electronic trade scenarios, offering both a graphical representation and a formal basis for the verification of various properties.

In order to make the scenarios adaptable, scenario components are broken down into reusable component parts, which can be flexibly reassembled to meet the needs of a wide variety of situations. The contract reusability is an issue in many approaches e.g. [MB98, GM00, KGV00]. The idea to identify reusable components in contracts is appealing but, as it can be observed from this survey, no significant progress has been made. The reason for this is the huge diversity in the contracts and their clauses. Some preliminary work on this topic has been done at Twene University (CrossFlow Project) as well.

2.9 COSMOS

The COSMOS (Common Open Service Market for SMEs) [CSP99] project is presented in [GBW98]. The project aim is an Internet-based electronic contracting service that facilitates business transaction processes. The e-contracts are considered as a solution for reducing the high transaction costs in the standard contractual process. Three groups of transaction costs are described; information costs, negotiation costs, and execution costs.

The COSMOS project aims at providing an infrastructure that allows the integration of all phases of e-contracting, based on object-oriented Internet technology. The project's goal is to establish a technology to create complex contracts in an easy way and to support their semi-automated filling in. Further on, COSMOS aims at supporting the negotiation and execution phases by letting the constructed contract actively influence the processing of itself. Due to the integrated, semi-automated construction of the contract, the COSMOS system should be able to consistently include execution definitions that can automatically drive the contract's fulfilment. This approach resembles the CrossFlow work [KGV00], where the workflow definition and the enactment clauses in the contract have this function.

The COSMOS electronic commerce architecture is composed of an online catalogue, brokers, contract negotiation support, signing support, and contract execution support. The project uses the *CORBA Business Objects Architecture* (BOA). Voyager, a Java-based ORB that provides mobility of objects and is compatible with CORBA, is chosen as

implementation platform for COSMOS. Further on, a Contract Object Model is described (see figure 2.14). The contract model distinguishes several parts:

The *Who* part: Parties, Persons, and Signatures are related to the participants of the contract.

The *What* part is the subject of the contract. It covers all obligations of the involved parties. An important feature of the obligation is a list of QoS attributes.

The *How* part defines execution details for the obligations: when and which services to be delivered; what is the deadline; which clause will apply when a party does not observe its obligation. This part is used to derive a workflow that defines causal relationships, data transfers, delays and deadlines, and the final termination of the execution phase.

The *Legal clauses* form the fourth part of a contract.

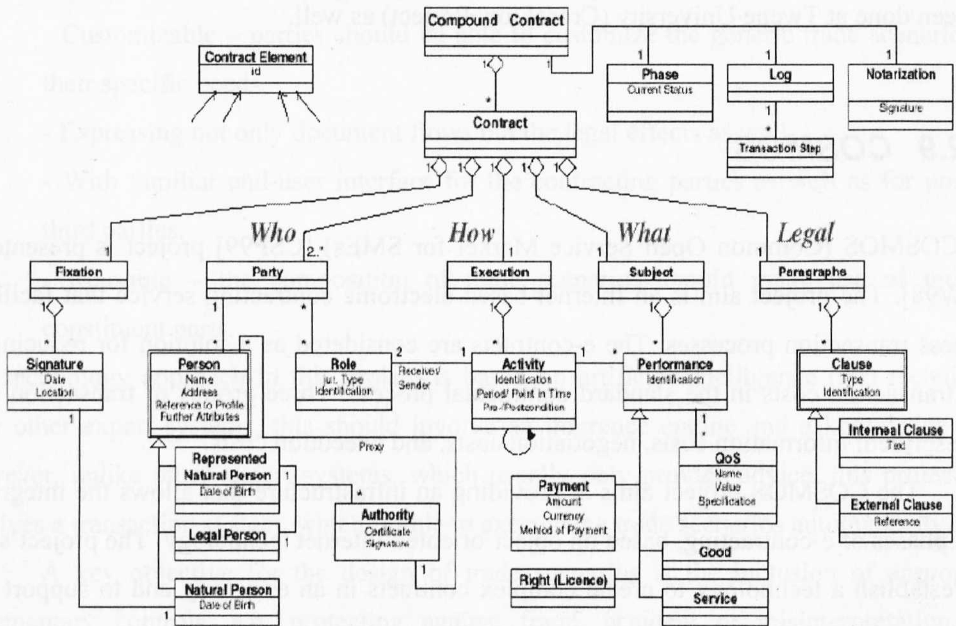


Fig.2.14. COSMOS contract model [GBW98]

Once a contract has been filled and signed, it becomes executable. The information gathered within the contract during contract filling phases directly allows deriving a workflow to execute a contract. Thus, a graphical representation of a workflow based on Petri-nets can be generated from the contract. The workflow can be used only as a Petri-net interpreter that has adapters to different workflow environments. For companies not having established their own internal workflow system, COSMOS additionally includes a self-installing workflow environment.

According to this paper, the attempt to cover the full semantics of a contract by building a "contracting expert system" is considered a dead end, since the expert system overhead i.e. the complexity of the system is expected to be too high.

2.10 Trade intermediaries

In [YAW98] the authors discuss the role of the trade intermediaries (e.g. retailers) in electronic commerce. The need for a trade intermediary is often dismissed in the direct supplier to consumer electronic commerce transaction model. For example, an e-commerce system has little need for distributional intermediaries.

The problem of electronic markets that is depicted in this paper is the uncertainty about product quality. One of the primary reasons why a market fails is the asymmetric information: what a seller knows is different from what a buyer knows. The authors suggest electronic commerce intermediaries to act as quality guarantors but without incurring high transaction costs. The paper answers to the problem of how to keep the costs low for intermediaries when the quality cannot be observed as in the physical world. A possible solution is a just-in-time purchasing system. The key element of this system is the open-ended contract with suppliers whose deliveries are not inspected. The contract can be terminated if the intermediary encounters many instances of low quality.

The reward for high quality is the continued business relationship with the manufacturer. The rest of the paper discusses the role of micro-payments in the electronic commerce, which is mostly relevant to B2C transactions.

2.11 Legal contracts as processes

Work that considers state representation of contracts is introduced in [DDM01] and [D00]. In [D00] an informal schematic notation for electronic contracts is introduced. It can be used to summarize the structure of agreements as collections of interrelated obligations. However it seems as though formal semantics had not been developed for the notation. In [D00], the authors present a simple architecture for an e-market where a controller agent is used to undertake the resolution of possible disputes between parties to an agreement. The controller may hold a representation of a contract in a model language, which implicitly defines state spaces. Also the representation is accessible to each party so that each party knows what it is supposed to do, and what to expect from its counter party.

The controller in this architecture acts as a judge, using information from the contract, and other sources such as advisors for the resolution of disputes. This is a different line of research to ours, as we concentrate explicitly on using Finite State Machines to represent contracts, and enforce them. Any disputes that arise in our case are not currently addressed by our research.

2.12 Event-Trigger-Rules

Work done at the University of Florida proposes an approach, called the Event-Trigger-Rule (ETR) paradigm, which is motivated by the need for rule based processing capabilities in the distributed environment of electronic commerce enterprises [SL01].

The ETR paradigm is a generalisation of the ECA (Event Condition Action) approach where the event specification and conditions and actions of the rule are specified as separate entities. Specifying a trigger then associates the event and rule together into a policy. This is in contrast to the ECA rule specification approach, where the event specification, associated conditions and actions be combined into a single rule.

In the ETR approach, events can be classified into 3 types – method associated events, explicit events and timer events. Method events are associated with a particular method invocation and can be raised either before, after or on-commit of the method. This distinction is referred to as the coupling mode of the event.
Each of these coupling modes raises synchronous events that will cause a rule to be evaluated before execution of the program continues. Additional coupling modes are instead-of (raises a synchronous event that allows the rule to replace the method invocation) and decoupled (raises an asynchronous event).

Explicit events are those raised by the application during execution and timer events are those associated with a particular time of interest. Next is a method associated event specification. Specification of a method associated event:

```
IN InventoryManager
EVENT update_quantity_event(String item, int quantity)
TYPE METHOD
COUPLING_MODE BEFORE
OPERATION UpdateQuantity(String item, int quantity)
```

A rule specifies some operations that should be performed if certain conditions apply. The conditional part of an ETR rule is defined as a guarded expression, where the guard is used to control evaluation of the conditional expression. This allows the entire rule to be skipped if any part of the guard expression evaluated to false, thus avoiding potential exception conditions (e.g. if required variables are not initialised).

Additionally, the rule specifies an action block (cf. a 'then' block) and an alternative action block (cf. an 'else' block). The complete syntax of a rule specification can be seen here: Syntax of the rule specification:

RULE rule_name(parameter list)
[RETURNS return_type]
[DESCRIPTION description_text]
[TYPE DYNAMIC/STATIC]
[STATE ACTIVE/SUSPENDED]
[RULEVAR rule variable declarations]
[CONDITION guarded expression]
[ACTION operation block]
[ALTACTION operation block]
[EXCEPTION exception and handling block]

When specifying a rule, it is possible to define local variables using the RULEVAR clause and also handle errors using the EXCEPTION clause. The STATE clause specifies if the rule will be active or suspended after its definition. A suspended rule will not be triggered until it is made active. The specification syntax also provides optimisation hints to the runtime environment using the TYPE clause. A dynamic rule can be changed at runtime whereas a static rule is less likely to be changed. This information is used when generating the runtime representation of the rule to provide optimal performance. The final component of the ETR approach is the trigger. Triggers are used to specify which event(s) causes the processing of a particular rule. Syntax of the trigger specification:

TRIGGER trigger_name(parameter list)
TRIGGEREVENT set of event connected by OR
[EVENTHISTORY event expression]
RULESTRUC set of rules
[RETURNS return_type: rule_in_RULESTRUC]

The TRIGGEREVENT clause is used to specify the set of events, combined using an OR connective, that will cause the rule(s) specified in RULESTRUC to be evaluated. The event specification can be augmented using the EVENTHISTORY clause to define other event expressions that need to have occurred prior to the one defined in the TRIGGEREVENT clause. When specifying the rules to be triggered in the RULESTRUC clause, it is possible to combine several rules using one of 4 constructs: sequential (rules are triggered one after the other), parallel (rules are triggered concurrently), AND-synchronised (all members of a set of rules must complete evaluation before another, specified, rule is triggered), and OR-synchronised (any two members of a set of rules must complete evaluation before another, specified, rule is triggered).

The literature that discusses the ETR approach presents many applications of this technique. These include the development of a knowledge management network [L00] and in a dynamic business process management service described in [SL01].

Based on its similarity to the ECA rule approaches like PDL (Policy Description Language) [LN99], it is easy to see how the ETR approach could be used to specify obligation policies in a distributed system. However, because of the manner in which events are defined and the ability to associate them to method invocations, it is also possible to specify authorisation policies, albeit less succinctly, using this notation. Additionally, by separating the event specifications from the rules, the ETR approach allows the user to reuse the events in multiple triggers and thus associate them with different rules as necessary. Despite the ability to specify different types of policy, and reuse parts of the specification in multiple rules, this approach does not support other useful features like policy extension (defining policies that inherit features from some parent policy) or policy groupings (organising policies that relate to the same activity together).

2.13 Ponder Policy Specification Language

Of relevance to contract monitoring and enforcement is the Ponder Policy Specification Language [DDLS01]. Ponder is an object-oriented declarative language for specifying management and security policies for distributed systems or contractual service level agreements between business partners. It can specify, monitor and enforce what actions (operations on objects) are permitted within a system, who can invoke the actions and under which conditions. It specifies policies in terms of obligations, permissions and prohibitions and provides means for defining roles and relationships. Ponder comes with a toolkit for editing, compiling and managing policies, that can be downloaded from its Web page at the Department of Computer Science of the Imperial College in London [P02].

To detect and prevent policy conflicts such as conflict for a given resource or overlapping of duties, Ponder's notation permits the specification of semantic constraints that limit the applicability of a given policy in accordance with the person playing the role, time, or state of the system.

Key concepts of the language include domains to group the object to which policies apply, roles to group policies relating to a position in an organisation [LS97], relationships to define interactions between roles and management structures to define a configuration of roles and relationships pertaining to an organisational unit such as a department.

Ponder Domains:

Domains provide a means of grouping objects to which policies apply and can be used to partition the objects in a large system according to geographical boundaries, object type, responsibility and authority or for the convenience of human managers. Membership of a

domain is explicit and not defined in terms of a predicate on object attributes. A domain does not encapsulate the objects it contains but merely holds references to objects. A domain is thus very similar in concept to a file system directory but may hold references to any type of object, including a person. A domain, which is a member of another domain, is called a **sub-domain** of the parent domain. A sub-domain is not a subset of the parent domain, in that an object included in a sub-domain is not a *direct* member of the parent domain, but is an *indirect* member, c.f., a file in a sub-directory is not a direct member of a parent directory. An object or sub-domain may be a member of multiple parent domains i.e. domains can overlap. An advantage of specifying policy scope in terms of domains is that objects can be added and removed from the domains to which policies apply without having to change the policies.

Ponder primitive policies:

Authorisation policies define what activities a member of the subject domain can perform on the set of objects in the target domain. These are essentially access control policies, to protect resources and services from unauthorized access. A positive authorisation policy defines the actions that subjects are permitted to perform on target objects. A negative authorisation policy specifies the actions that subjects are forbidden to perform on target objects.

The language provides reuse by supporting the definition of policy types to which any policy element can be passed as a formal parameter. Multiple instances can then be created and tailored for the specific environment by passing actual parameters as shown in the following example:

```
type auth+ PolicyOpsT (subject s, target < PolicyT> t)
        {
        action load(), remove(), enable(), disable() ;
        }
        inst auth+ switchPolicyOps=PolicyOpsT(/NetworkAdmins, Nregion/switches);
        inst auth+ routersPolicyOps=PolicyOpsT(/QoSAdmins, /Nregion/routers);
```

Which means that the two policy instances created from a *PolicyOpsT* type allow members of /NetworkAdmins and /QoSAdmins (subjects) to load, remove, enable or disable objects of type PolicyT within the /Nregion/switches and /Nregion/routers domains (targets) respectively.

Policies can also be declared directly without using a type as shown in the negative authorisation policy next, which indicates the use of a time-based constraint to limit the applicability of the policy:

```
inst auth– /negativeAuth/testRouters {
subject /testEngineers/trainee ;
action performance_test() ;
target <routerT> /routers ;
when time.between ("0900", "1700")
}
```

Specifies; trainee test engineers are forbidden to perform performance tests on routers between the hours of 0900 and 1700. The policy is stored within the /negativeAuth domain.

Ponder also supports a number of other basic policies for specifying security policy: *Information filtering* policy can be used to transform input or output parameters in an interaction. For example, a location service might only permit access to detailed location information, such as a person is in a specific room, to users within the department. External users can only determine whether a person is at work or not. *Delegation* policy permits subjects to grant privileges, which they possess (due to an existing authorisation policy), to grantees to perform an action on their behalf e.g., passing read rights to a printer spooler in order to print a file. *Refrain* policies, define the actions that subjects must refrain from performing (must not perform) on target objects even though they may actually be permitted to perform the action. Refrain policies act as restraints on the actions that subjects perform and are implemented by subjects. See [DD01] for more details and examples of these policies.

Obligation policies are event-triggered condition-action rules, and define the activities subjects (human or automated manager components) must perform on objects in the target domain.

Events can be simple, i.e. an internal timer event, or an external event notified by monitoring service components e.g. a temperature exceeding a threshold or a component failing. Composite events can be specified using event composition operators.

```
inst oblig loginFailure {
on          3*loginfail(userid) ;
subject     s = /NRegion/SecAdmin ;
target      <userT> t = /NRegion/users ^ {userid} ;
do          t.disable() -> s.log(userid) ;
}
```

This policy is triggered by 3 consecutive loginfail events with the same userid. The NRegion security administrator (SecAdmin) disables the user with userid in the /NRegion/users domain and then logs the failed userid by means of a local operation performed in the SecAdmin object. The '->' operator is used to separate a sequence of actions in an obligation policy. Names are assigned to both the subject and the target. They can then be reused within the policy. In this example we use them to prefix the actions in order to indicate whether the action is on the interface of the target or local to the subject.

Ponder Composite policies:

Ponder composite policies facilitate policy management in large, complex enterprises. They provide the ability to group policies and structure them to reflect organisational structure, preserve the natural way system administrators operate or simply provide reusability of common definitions. This simplifies the task of policy administrators.

Conflict analysis

In [DBSL02], Damianou et al discuss the issue of conflict resolution within policies. This is of relevance to our efforts outlined in this thesis on validation of rights and obligations for the purpose of removing ambiguities within contracts.

[DBSL02], discusses different types of conflicts and presents strategies for resolving them. A classification of policy conflicts is presented in [LS99], which discussed both modality conflicts and application specific conflicts. Modality conflicts can be categorised into three distinct types:

> **Authorisation conflicts** arise when a positive and a negative authorisation policy is defined for the overlapping subjects, targets and actions.
>
> **Obligation conflicts** arise when one policy obliges a subject to perform a given action whilst at the same time another policy forbids the action from being performed. In the context of Ponder, this situation would arise if an obligation and a refrain policy were defined on overlapping subjects and targets with identical actions.
>
> **Unauthorised obligation conflicts** arises when a subject is obliged to perform an action that it does not have the authorisation to do. In a system with a default negative authorisation policy in which actions have to be explicitly authorised, this could occur if an obligation policy is defined without an associated authorisation policy.

Application specific conflicts are those that arise because of constraints defined for the particular application in which the policies are being used. For example, a system that enforces the principle of separation of duties would define a conflict if the same person who submits an expense report is also allowed to approve it.

[JS97] identifies that conflicts can be either static or dynamic. The distinction is that analysing the syntax of a policy statement can identify static conflicts. These conflicts will occur irrespective of the state of the system enforcing the policies – this is often the case for simple modality conflicts. Dynamic conflicts are those that occur at run-time and arise because a particular state of the system results in a conflict. These are harder to detect in advance given that it is necessary to analyse the system in all possible states to do so.

[S97] proposes that a conflict, once detected could be handled in one of three ways. The most obvious and simplest one is for the system to declare an error condition whenever a conflict arises. However, this solution is not particularly interesting since it does not allow for the system to automatically recover from the conflicting scenario. Other solutions are to allow the positive policy to override; or to let the negative policy override. The latter strategy is adopting an approach of 'do no harm', based on the assumption that the negative policy (i.e. the one that prevents an action being performed) has a more benign effect on the system than its conflicting counterpart. As would be expected, the positive policy override strategy is the exact converse of the negative override approach described.

In addition to the negative and positive override strategies mentioned above, [LS99] also identifies some alternatives. One approach suggested is to assign explicit priorities to every policy. This way when a conflict arises, the agent enforcing the policy could simply compare the priority values and enforce the policy that has the highest priority. However, this approach could easily lead to inconsistent behaviour of the system if, as is common in distributed systems, multiple people are responsible for defining policies and assigning their priorities. Other strategies suggested include giving priority to the policy that is 'closest' to the managed object; or using the specificity of the policy definition to determine the priority.

Work done by Chomicki and Lobo [CL00] describes how conflicts can arise between ECA rules and action constraints defined in the policy Description Language (PDL). Here, a policy monitor is defined to detect conflicts between the ECA rules and any action constraints. In order to resolve the conflict, the monitor will either choose to ignore certain events, thus preventing the ECA rule from activating and causing the conflict; or will cancel any actions that are specified in an action constraint. The latter scenario is an example of a negative policy override strategy.

Progress has been made in dealing with policy conflicts within ponder, however significant challenges remain to be addressed [DBSL02]. In particular, how can one detect conflicts when arbitrary conditions restrict the applicability of the policies? Sometimes, it is possible to compare restrictions placed by the constraints. For example, it is possible to detect if two time intervals overlap or if the policies apply when subjects are in different states e.g., active or standby. However, the problem remains unsolved in the general case. Other challenges concern the different levels of abstraction at which policy is specified. Also Conflicts between organisational goals will inevitably lead to conflicts between the policies derived from these goals. Some policies will trigger complex management procedures, which require the execution of actions that may be specified as part of different policies. This renders the task of ensuring the consistency of policy specification much more complex.

2.14 E-Commerce Frameworks

Here we list a number of e-commerce frameworks, which contain elements such as modelling methods, supporting tools, standards, software and system architectures etc. They are regarded important to e-contracting because they each include contracting as one of their major modules.

2.14.1 ebXML

The United Nations body for Trade Facilitation and Electronic Business (UN/CEFACT) and the Organization for the Advancement of Structured Information Standards (OASIS), have joined forces to initiate a worldwide project to standardize XML business specifications. UN/CEFACT and OASIS have established the Electronic Business XML initiative to develop a technical framework that will enable XML to be utilized in a consistent manner for the exchange of all electronic business data. Industry groups currently working on XML specifications have been invited to participate in the 18-month project. A primary objective of ebXML is to lower the barrier of entry to electronic business in order to facilitate trade, particularly with respect to small- and medium-sized enterprises (SMEs) and developing nations. The first ebXML Initiative Technical Specifications has been released for public comment. The ebXML Requirements Specification defines specific technical infrastructure requirements [AG01]. More information can be found at [EBXML], [UNCE], and [OASIS].

2.14.2 BizTalk

BizTalk is an industry initiative started by Microsoft and supported by a wide range of organizations, from technology vendors like SAP, CommerceOne, and Ariba to technology users like BASDA. It includes a design framework for implementing an XML schema and a set of XML tags used in messages sent between applications. It assumes that applications are distinct entities, and application integration takes place using a loosely coupled approach to pass messages. The two applications simply need to be able to format, transmit, receive and process a standardized XML message. Through the BizTalk web site one can locate, manage, learn, share information about and publish XML, XSL and information models and business processes supported by applications that support the BizTalk Framework [BIZ].

2.14.3 Web Services

A Web service is an interface that describes a collection of operations that are network accessible through standardized XML messaging. A Web service is described using a standard, formal XML notion, called its service description that provides all of the details necessary to interact with the service, including message formats (that detail the operations),

transport protocols, and location. Web service descriptions are expressed in the Web Services Definition Language or WSDL.

Web Services Technology, built upon existing and emerging standards such as HTTP (Hyper Text Transfer Protocol), XML (Extensible Markup Language), SOAP (Simple Object Access Protocol), WSDL (Web Services definition Language), and UDDI (Universal Description Discovery and Integration), is speeding the development of Business to Business (B2B) applications, and thus accelerating the expansion of the Internet. [GGKS02].

Web services technologies provide a programming model that accelerates application integration inside and outside the enterprise. Because Web services are easily applied as a wrapping technology around existing applications and IT assets, new solutions can be deployed quickly and recomposed to address new opportunities.

As adoption of Web services accelerates, the numbers of services will increase, fostering development of more dynamic models of just-in-time application and business integration over the Internet.

Currently Web Services only provide for simple communication between computer software, they do not support business interactions. Efforts are underway to enable such interactions however. BPML (Business Process Execution Language) [BPML] is a notion for specifying business process behaviour based on web services.

Processes in BPML export and import functionality by using web services interfaces exclusively. BPEL4WS (Business Process Execution Language for Web Services) provides a language for the formal specification of business processes and business interaction protocols. By doing so, it extends the Web Services interaction model and enables it to support business transactions. BPEL4WS defines an interoperable integration model that should facilitate the expansion of automated process integration in both the intra-corporate and the business-to-business spaces [BPML].

Related references: [PROT] [SOAP] [WSDL] [IBMW].

2.14.4 GRID

Grid computing is an evolving area of computing, where standards and technology are still being developed to enable this new paradigm. It is a form of distributed computing that involves coordinating and sharing computing, application, data, storage, or network resources across dynamic and geographically dispersed organizations. Grid technologies promise to change the way organizations tackle complex computational problems [GRID].

Research and development efforts within the Grid community have produced protocols, services, and tools that address the challenges that arise when we seek to build scalable Virtual Organisations. These technologies include security solutions that support management of credentials and policies when computations span multiple institutions;

resource management protocols and services that support secure remote access to computing and data resources and the co allocation of multiple resources; information query protocols and services that provide configuration and status information about resources, organizations, and services; and data management services that locate and transport datasets between storage systems and applications [FKT01].

OGSA (Open Grid Services Architecture) [OGSA], is an alignment of Grid technologies, and Web Services technologies. This architecture is still evolving and uses Web Services Description Language (WSDL) to achieve self-describing, discoverable services and interoperable protocols, with extensions to support multiple coordinated interfaces and change management. The aim is a standards-based distributed service system that supports the creation of the sophisticated distributed services required in modern enterprise and inter-organisational computing environments [FKNT02].

2.14.5 eCo Framework

The eCo framework is a CommerceNet project that focuses on the integration of three e-commerce services. These services are: an integration of multiple database types with multiple data constructs and data libraries; trusted open registries; and agent mediated buying. The intent is that these core services will provide interoperability between many commerce services and will serve as a foundation to operate web based trading communities [ECO].

2.15 Discussion

Electronic contracts have drawn the interest of many research groups and projects such as ODP-RM, CrossFlow, COYOTE, Queensland University, LGI, InterPocs, COSMSOS, TINA, Ponder...etc.

Of the projects discussed in the previous sections, those that have mainly drawn our attention are Milosovic et al's research on electronic contracting (Section 2.4), and Naftaly Minsky's Law Governed Interaction (Section 2.7). Specifically of interest to us are the software tool Milosovic proposes for the contract validation process, and more elaborative details on the operation of the mediator controller Cxy in Minsky's LGI infrastructure.

For the purpose of modelling business transactions that are derived from contracts, none of the approaches above use finite state machines, some such as InterPocs and COSMOS make use of Petri-nets, however the majority of them rely on elaborate logical notations that include temporal constraints and role players in their parameters. The expectation is that these notations should be able to specify arbitrarily complex business contracts and detect all kind of inconsistencies during the contract validation process. In most

of the works above (Queensland, St Gallen, Ponder...etc), the contract infrastructure is intended to be embedded within the infrastructure of the organisation. The expectation is that the executable contract will not only be able to monitor and enforce the agreement between the parties, but also will be able to take into account the organisations' internal policies, trying to ensure they do not conflict with the clauses of the contract.

This generality is certainly desirable; however, because of the complexity of the problem it might be rather ambitious. We believe that a modular approach is more realistic for detecting contract conflicts and ambiguities. For that to be possible, we need to be able to identify and isolate the different sources of possible inconsistencies in business contracts.

In our model, we make a clear distinction between an organisation's internal policies, and the external policies that it may have signed on within the context of a contract. From this perspective, we can identify two fairly independent sources of contract inconsistencies:

- Internal enterprise policies conflicting with contractual clauses.
- Inconsistencies in the clauses of the contract.

It is our view that these two issues should be treated separately rather than encumbering a contract model with excessive notation (details, concepts and information) that might be extremely difficult to validate. Such a separation is not considered in the work discussed above. In this thesis we address only the second issue, that is, we are concerned only with the cooperative behaviour of business enterprises and not their internal structure.

In our business model each contracting enterprise has the privilege and responsibility of verifying that its internal policies do not conflict with the clauses of the contract. Similarly, each enterprise exercises its independence to choose the roles players that would invoke operations on the contract and provide them with a proper contract role player certificate (a cryptographic key for example). Consequently, it is the responsibility of each enterprise to prevent inconsistencies with role players such as duty overlapping, duty separation, etc.

We intentionally leave the notion of role players out of our discussion. However, we assume they are authenticated by the contract management system before they are allowed to perform operations of the FSMs. It can be argued that our FSM model lacks expressiveness in comparison with the related works discussed above. However we do gain in simplicity. Thanks to this simplicity we can use standard of-the-shelf model checkers like Spin [SP03] to validate general safety and liveness properties of contracts, relatively easily.

In our work, contracts are conceptually located between the interacting parties and are meant to drive the execution of inter-enterprise business processes.

Business processes vary in complexity from rather small such as the purchase of a book to the rather complex such as the booking of a package holiday that could involve any number of organizations, which have many agreements between them

We assume that a complex business process can always be decomposed into two or more business processes of lower complexity that perform specific and individual activities. This decomposition can be conducted several times until the complexity of the resulting sub-processes is manageable [MSS03].

This decomposition approach is of great relevance for the implementation of x-contracts. With x-contracts the interaction between the business partners can be thought as taking place through individual sub-processes that are regulated by individual sub-contracts. Naturally, each sub-contract contains only the rights and obligations to regulate the activities involved by the particular sub-process. For example, two business partners might have a contract that contains two sub-contracts: one for processing purchase orders for perishable goods and a different sub-contract for tinned food. To execute a complete business contract a parent contract is given the information and the power to create, coordinate and terminate one or more instances of the same or different sub-contracts as needed. When an instance of a sub-process is instantiated or terminated by the business partners, its corresponding electronic sub-contract is instantiated or terminated by the parent contract.

An x-contract will in many cases, be simple enough so that it involves only one main process, i.e. only one sub-contract will exists, so in fact the contract and its sub-contract will be the same. In this thesis we are concerned with the execution of sub-contracts only, whether this means the execution and monitoring of single process "simple" contracts, or whether it means executing and monitoring individual processes within a bigger and more complex contract that includes more than one sub-contract. We can briefly mention that within "complex" contracts, the parent contract can be realised as a workflow script that manages the set of sub-x-contracts that represent the set of sub-contracts that compose the whole contract. In the rest of this thesis we refer to our sub-processes, sub-contracts, and sub-x-contracts simply as processes, contracts, and x-contracts, respectively.

Blank Page

Chapter Three

Electronic Contracts as Finite State Machines

In this chapter, we show how contracts can be converted into executable electronic contracts through a process in which finite state machines are used to describe, monitor, and enforce the clauses stipulated within a contract.

We first describe the meaning of x-contracts, and show how the clauses that form a contract can be split into rights and obligations.

After this, we introduce finite state machines and illustrate how x-contracts, like communication protocols, can be abstracted by finite state machines, and importantly we look at how a finite state machine can express the rights and obligations of an x-contract. We illustrate our ideas with the aid of a simple example of an ambiguous contract. We show how the ambiguous contract is modelled into finite state machines. Finally after ambiguities are removed from the finite state machine contract model, we look at how finite state machines can be used to actively monitor and enforce the clauses of the contract.

As shown in Chapter 4, one of the major advantages of using FSMs is the facilities they offer to analyse the correctness properties of contracts using validation tools such as Spin.

3.1 Contracts and X-Contracts

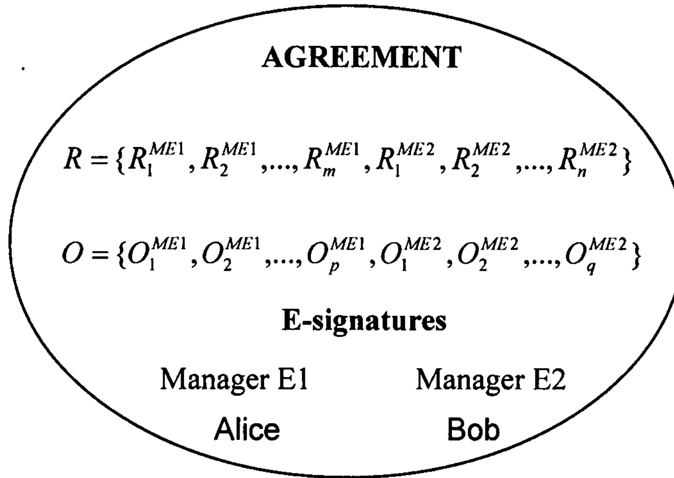
A *contract* can be defined as a paper document that stipulates that the signatories (two or more) agree to observe the clauses stipulated in the document. Each entry in the document is called a *term* or a *clause*.

An x-*contract* is an executable electronic version of a conventional contract. It is an electronic document that monitors that the signing entities observe clauses stipulated in the document.

As can be seen from the definitions of conventional and electronic contracts, the main idea behind conventional and electronic contracts is the same: the two of them regulate the interaction between two or more trading parties. However, in spite of the close similarities, there is a crucial difference between the two kinds of contract. A conventional contract is human oriented, whereas an x-contract is computer-oriented. Consequently, the former tolerates ambiguities, the latter does not.

3.1.1 Rights and obligations

The clauses of a contract stipulate how the signing parties are expected to behave. In other words, they list the rights and obligations of each signing party. It should be possible to precisely extract a set of rights and a set of obligations from the clauses of a given contract. Therefore, a contract with two contracting parties can be represented as shown in figure 3.1.



E1,E2—Enterprises, R_i—Right, O_i—Obligation
ME1—Manager of E1, ME2—Manager of E2

Fig. 3.1. Abstraction of the main elements of a contract.

For the sake of simplicity, we will discuss an example with only two contracting parties. However, all our concepts, and models, can be generalised to $n \geq 2$ parties as long as n is finite.

Let us assume that in figure 3.1, E1 is managed by Alice and that E1 is interested in purchasing some items from E2. Similarly, Bob is the manager of E2 which is an enterprise interested in supplying items to E1. The contract shown in this figure has been signed by Alice as the manager of enterprise E1 after agreeing with Bob that she will observe *m* rights, $R=\{R_1^{ME1},R_2^{ME1},...,R_m^{ME1}\}$ and *p* obligations, $O=\{O_1^{ME1},O_2^{ME1},...,O_p^{ME1}\}$. On the other hand, Bob signed that he, as the manager of enterprise E2, would observe *n* rights, $R=\{R_1^{ME2},R_2^{ME2},...,R_n^{ME2}\}$ and *q* obligations, $O=\{O_1^{ME2},O_2^{ME2},...,O_q^{ME2}\}$.

We define a right as an authorization to do something. Because it is only an authorization, a right may or may not be exercised. In the context of the execution of an x-contract, a right is an authorization to perform an operation that will affect the behaviour of the execution of the x-contract. For example, the signed contract of figure 3.1 can stipulate

that Alice, as a manager of E1, has the right to send an offer to sell to Bob, the manager of E2. Because this is a right, it is up to Alice to send or not to send the offer to Bob; Bob will not be disappointed if he does not receive the offer.

An obligation can be defined as a duty that must be performed. In the context of the execution of an x-contract, an obligation is a duty to perform an operation that will affect the behaviour of the execution of the x-contract. A failure to perform such a duty means a breach of the x-contract. For example, the text of the x-contract shown in figure 3.1 might stipulate that upon receiving an offer to sell from Alice, Bob has the obligation to reply to her with an OfferAccepted or OfferRejected message.

The execution of a right or an obligation such as SendOfferAccepted or SendOfferRejected will, at a lower level of abstraction, demand access to one or more objects such as files, databases and printers. A question that arises here is whether Alice and Bob have the right to access the objects affected by their operations. This is an issue of authentication and can be left out of the discussion while we talk at the level of rights and obligations, and will be discussed in detail in Chapter 6. At this level we can assume that the persons that execute the x-contract are granted permission to access the objects they need.

Note that in our x-contract each right and obligation is given a name; naming rights and obligations is crucial in our understanding of x-contracts. Being able to name each right and obligation individually means that we can identify, and monitor each of them at run-time, that is, when the contract is enacted.

3.2 Finite State Machines

A Finite State Machine (FSM) is a widely used and well known model for protocol specification. Since its introduction in the 1950s it has been used for modelling a great number of systems. Its analytical power and the ease with which the model can be loaded into a computer and manipulated automatically with the help of software tools makes this method attractive for modelling protocols. Similarly, the graphical nature of this model makes it easy to read and understand the different stages that the protocol goes through during its execution.

Formally, a finite state machine M is defined as the quintuple $[S, I, Z, \delta, \lambda]$, where $S = \{s_1, s_2, ..., s_m\}$, $I = \{i_1, i_2, ..., i_n\}$ and $Z = \{z_1, z_2, ..., z_p\}$ are finite nonempty sets of states, input symbols and output symbols, respectively. $\delta : S \times I \rightarrow S$ is the transition function and $\lambda : S \times I \rightarrow Z$ is the output function.

Informally, M describes an abstract system that stays in a given state until it receives an external stimulus. When such stimulus is received, the system reacts by doing something (for example, sending an output signal) and then moves to a different state. Note that *do*

something might mean do nothing in some circumstances and that the new state is not necessarily different from the previous. The behaviour of this abstract system is deterministic. The quintuple $[S, I, Z, \delta, \lambda]$ unambiguously defines what to do and where to go next.

Because of their high level of abstraction, FSMs are used to describe and model a great variety of systems. In particular, the computer science community has gained a great deal of experience in the use of FSMs for describing communication protocols, and built several tools for validating such protocols. For example, Spin [SP03], and LTSA [LTSA99] are well known protocol validators.

3.3 Representing Contracts as Finite State Machines

We have introduced communication protocols into the discussion about x-contracts for a valid reason: we argue that from the point of view of the interaction and synchronisation between the parties involved, x-contracts are equivalent to communication protocols. We claim that x-contracts, as communication protocols, can be precisely abstracted by FSMs. The advantage of looking at contracts as FSMs is that we can put into practice all the existing machinery that was originally developed for studying communication protocols. For instance, we can resort to Spin to validate an x-contract before converting it into the actual computer program that will enact it. The goal of a validation process is to analyze what is known as the correctness properties of the system. In other words, the essence of the validation is to discover, at an early stage, whether the execution of an x-contract takes the contracting parties into unacceptable situations. Among other things, validating the FSM model of an x-contract should reveal the existence of states (conditions in the x-contract) that are not reachable, that is, states for which there is no path from the initial state. If one of these unreachable states represents the receipt of the goods the situation would be unacceptable and the contract would need to be re-written. In the same order, the validator should show that at some point, the two contracting parties reach a final state (contract deal for example) instead of being left in a transient state for ever. To mention another example, the validation should reveal whether the contract allows purchasers to receive goods before paying for them. FSM and contract validation will be looked at in more detail in the next chapter.

A question that naturally arises at this point is how the rights and obligations of a contract can be expressed in a FSM.

3.3.1 Mapping Contract Clauses into FSMs

At the level of rights and obligations an x-contract is often more easily understood as a set of FSMs, one for each contracting party. So, from our example in figure 3.1, we have one FSM for the purchaser and one FSM for the supplier, FSM_P, FSM_S respectively.

The physical location of each FSM is irrelevant to the functionality of the contract and is decided at the time of implementation. For the moment let us assume that FSM_P is located within Alice's enterprise and FSM_S is located within Bob's enterprise. To enact the x-contract these two FSMs must share a common communication channel to interact with each other, that is, the output of FSM_P is somehow connected to the input of FSM_s and vice versa. We will now discuss how the rights and obligations stipulated in an x-contract can be mapped into the FSMs.

It is easy to reason about the operations of an x-contract, with the following general syntax in mind:

if $\quad$ *event*$_1$ & *condition*$_q$ = *true*
perform $\quad$ *operation*$_1$ $\quad$ *and switch to state*$_1$
else if $\quad$ *event*$_2$ & *condition*$_q$ = *true*
perform $\quad$ *operation*$_2$ $\quad$ *and switch to state*$_2$
... ...
else if $\quad$ *event*$_m$ & *condition*$_q$ = *true*
perform $\quad$ *operation*$_m$ $\quad$ *and switch to state*$_m$

This syntax expresses the idea that, at some point an x-contract can be at any of n possible conditions ($condition_1$, $condition_2$,...,$condition_n$). If the x-contract is in a given $condition_q$ (for example, *WaitingForOffer*), there is a finite and well defined set of events ($event_1$, $event_2$, ...,$event_m$) that can affect the future behaviour of the contract. The occurrence of $event_i$ determines what objects (variables, files, database, etc.) within the system change their values, that is, the event determines to which new condition the systems switches. Similarly, there is a finite and well defined set of operations ($operation_1$, $operation_2$,..., $operation_m$) that can be executed when the system is in $condition_q$. The $event_i$ determines the operation to be executed.

Bearing in mind the discussion in Section 3.2, we argue that in terms of FSMs, the set of conditions of the general syntax presented above can be mapped into the set S of states of a FSM. Similarly, the set of events can be mapped into the set I of input symbols of the FSM. In the same order, the set of operations can be mapped into the set Z of output symbols of the FSM. Finally, we can map the set of switches to the next condition into the transitional function δ, and the set of switches to the next output into the function λ. It is important to bear in mind that the operation *donothing* is a valid operation. In this discussion we represent it with the symbol ε. To summarize:

Conditions *are mapped into* ***S states****.*

Events *are mapped into* ***I inputs****.*

Operations *are mapped into* ***Z outputs****.*

Thus, in terms of FSMs, we can express the above syntax as shown in figure 3.2, where *e* and *o* stand for event and operation, respectively.

3.3.2 Description of a simple contract using FSMs

To show what rights and obligations look like, we will discuss a very simple example of a contract (See figure3.3) for offering and purchasing goods remotely, for example, over the Internet.

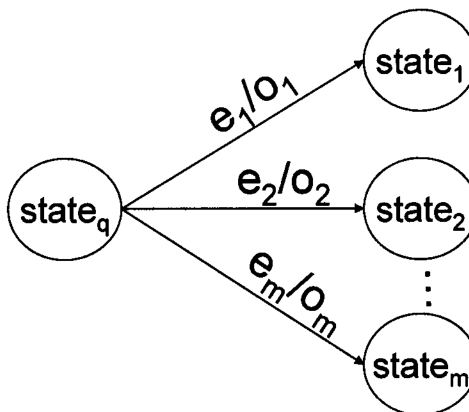


Fig. 3.2. Mapping of events, conditions and operations of a contract into a FSM state

This deed of agreement is entered into as of the effective date identified below.
Between
[Name] of [Address] (To be known as the (Supplier)), and [Name] of [Address] (To be knows as the (Purchaser)).
Whereas
(Supplier) desires to enter into an agreement to supply (Purchaser) with [Item].
Now it is hereby agreed that (Supplier) and (Purchaser) shall enter into an agreement subject to the following terms and conditions:
1. Definitions and Interpretations
1.1 Price, Dollars or $ is a reference to the currency of the [Country].
1.2 All information (purchase order, payment, notifications, etc.), is to be sent electronically.
1.3 This agreement is governed by [Country] law and the parties hereby agree to submit to the jurisdiction of the Courts of the [Country] with respect to this agreement.
2. Offer
2.1 The supplier may use his discretion to send offers to the purchaser.
2.3 The purchaser is entitled to accept or reject the offer, but he shall notify his decision to the supplier.
3. Commencement and completion
3.1 The contract shall start immediately upon signature.
3.2 The purchaser and the supplier shall terminate the x-contract immediately after reaching a deal for buying an item.
4. Disputes
*4.1 (Supplier) and (Purchaser) **shall** attempt to settle all disputes, claims or controversies arising under or in connection with the agreement through consultation and negotiations in good faith and a spirit of mutual cooperation.*
*4.2(Supplier) and (Purchaser) **shall** provide electronic evidences about breaches of the e-contract.*
4.3 This method of determination of any dispute is without prejudice to the right of any party to have the matter judicially determined by a [Country] Court of competent jurisdiction.
5 Amendment
*5.1 This agreement **may** only be amended in writing signed by or on behalf of both parties.*
E-SIGNATURES
In witness whereof (Supplier) and (Purchaser) have caused this agreement to be entered into by their duly authorized representatives as of the effective date written below.
Effective date of this agreement: [day] of [month] [year]

[E-signature]	*[E-signature]*
[Person]	*[Person]*
[Role]	*[Role]*

E-address for Notices:

[E-address]	*[E-address]*

Fig.3.3. Example contract between a Purchaser and a Supplier for the purchase of goods

As an attentive reader will notice, the contract has serious ambiguities; this will be discussed in Section 3.4.2. The contract which is signed by a purchaser and a supplier contains, among other data, the following clauses:

1 Offer
1.1 The supplier may use his discretion to send offers to the purchaser.
1.2 The purchaser is entitled to accept or reject the offer, but he shall notify his decision to the supplier.
2 Commencement and completion
2.1 The contract shall start immediately upon signature.
2.2 The purchaser and the supplier shall terminate the x-contract immediately after reaching a deal for buying an item.

From this English text contract clause we can extract the sets of rights and obligations for the purchaser and the supplier and express them in terms of operations for FSMs:

Purchaser's rights:
R_1^P : Accept offers.
R_2^P : Reject offers.
Purchaser's obligations:
O_1^P : Start the x-contract.
O_2^P : Reply to offers.
O_3^P : Terminate the x-contract.
Supplier's rights:
R_1^S : Send offers.
Supplier's obligations:
O_1^S : Start the x-contract.
O_2^S : Terminate the x-contract.

To be consistent with the notation in figure 3.1 we now specify the sets of rights and obligations: $R=\{R_1^P, R_2^P, R_1^S\}$ and $O=\{O_1^P, O_2^P, O_3^P, O_1^S, O_2^S\}$. We show how the sets R and O are mapped into FSMs in figure 3.4.

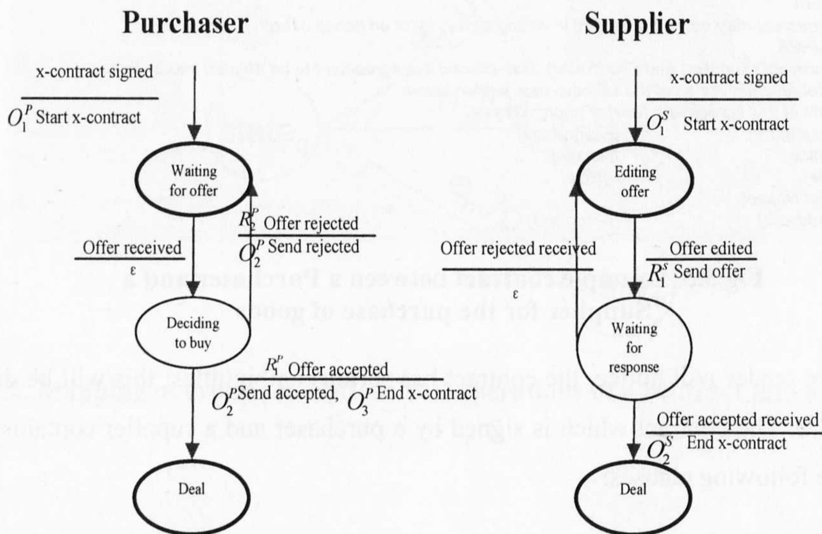


Fig. 3.4. FSM Representation of an ambiguous contract for the purchase of goods

As can be appreciated from figure 3.4, we have used, two FSMs to precisely describe the English text contract of our example. The elements of the sets of rights and obligations are also shown in the figure. However, as they are, the two FSMs of figure 3.4 only describe the behaviour of the two contracting parties; they do not monitor or enforce it.

3.4 Monitoring and enforcement of x-contracts

During the execution of an x-contract, rights and obligations are triggered by local and remote events. In this section we will show how two FSMs trigger rights and obligations on each other.

3.4.1 Invocation of rights and obligations

To reason about how the contractual rights and obligations can be monitored and enforced by a FSM, it is useful to look at the rights and obligations a contracting party has at a given state of the execution of the x-contract. In terms of FSMs, this is equivalent to looking at the set of operations that can be executed when the FSM of the contractual party is at $state_q$. It is useful to classify this set into two subsets: the subset of operations the owner of the FSM has the right to perform and the subset of operations that person has the obligation to perform, $\{o_1, o_2, ..., o_m\}$ and $\{o_{m+1}, o_{m+2}, ..., o_p\}$, respectively.

To illustrate how the rights and obligations are triggered we will examine figure 3.5. This figure shows a snapshot of the two FSMs that model an x-contract for the purchase and supply of e-goods.

Let us say, the execution of the x-contract at the purchaser's side, is at state $state_q$. As can be seen from the figure, the rights and obligations the purchaser has when his FSM is at $state_q$ can be mapped into the sets $\{o_1, o_2, ..., o_m\}$ and $\{o_{m+1}, o_{m+2}, ..., o_p\}$, respectively.

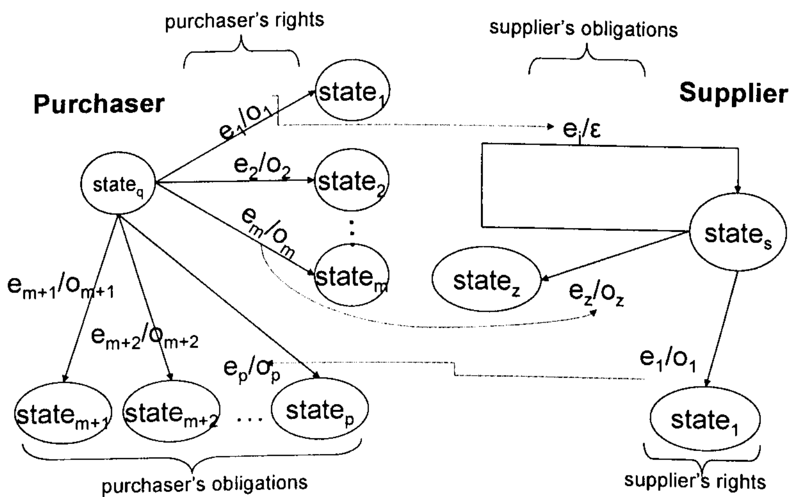


Fig. 3.5. Interaction of two FSMs by means of rights and obligations

Executing an operation from the subset $\{o_1, o_2, ..., o_m\}$ means exercising a right given by the x-contract. Since each operation o_i is paired to an event e_i, the operation o_i can be executed

only after the occurrence of e_i. How does event e_i occur? Event e_i can be triggered internally within the purchaser's enterprise or externally, say for example, within the supplier's enterprise. Since the execution of operation o_i is optional, the event of e_i might be deliberately triggered by the purchaser (for example, "I wish to send a purchase order"). Also, it can be the result of an unavoidable situation within the purchaser's enterprise and coded in the FSM (for example, the mainframe computer has crashed) or it can be triggered by a message received from the supplier (for example, "*would you like to buy this item?*").

Executing an operation o_{m+i} from subset $\{o_{m+1}, o_{m+2}, \ldots, o_p\}$ means complying with the contractual obligations the purchaser has when his FSM is at state$_q$. As with the rightful operations, the obligatory operations are paired to events which are triggered internally, or externally.

It is important to understand that exercising a right at one side of the contract might or might not have an effect at the other side. This depends on what the text of the contract stipulates. The execution of operation o_i at the purchaser's side might trigger a right, an obligation, or nothing at the supplier's side. By nothing we mean that the supplier's is not notified about the execution of the operation o_i at the purchaser's side.

Similarly, the execution of an obligatory operation o_{m+i} from the subset $\{o_{m+1}, o_{m+2}, \ldots, o_p\}$ might trigger a right, an obligation or nothing at the supplier's side.

The dashed line pointing from the pair e_1 / o_1 at the supplier's side to the pair e_p / o_p at the purchaser's side implies that at state$_s$ the supplier has the right to execute the operation o_1. The English text of the contract stipulates that the purchaser (being at state$_q$) has the obligation to execute operation o_p as a response.

Similarly, the dashed line pointing from e_m / o_m to e_z / o_z shows that at state$_q$ the purchaser has the right to execute the operation o_m. As a response to this operation, the supplier has the obligation to execute the operation o_z. The dashed line from e_1 / o_1 to e_1 / ε shows that the purchaser's has the right to execute the operation o_1. However, the execution of such operation demands nothing at the supplier's side.

To show how these ideas can be used in practice, we will apply them now to the example of an x-contract discussed in Section 3.3.2.

3.4.2 Description, monitoring and enforcement of an x-contract

In practice, it is likely that contracts will be written by lawyers and then passed on to technical people to convert the original English text into a computer program that will monitor and enforce what the contract stipulates.

From our own experience we have learnt that the first difficulty the technical person faces in this situation is the ambiguities that the English text contract is likely to have. The standard contract discussed in this section is not an exception. Although it looks correct at first glance, it has a serious ambiguity. The contract text does not specify the time for sending the offer. Neither does it specify the time for sending the notification about rejecting or accepting the offer. These two omissions render the English text contract difficult to convert into a useful x-contract. It is true that the x-contract can still be implemented and enacted but the purchaser's FSM will hang silently until the supplier decides to send an offer. If for some reason the supplier forgets to send his offer, the two FSMs will hang silently forever or until the purchaser or the supplier use another channel (a telephone, for example) to investigate the problem. Telephone calls are intensively used for clarifying situations in conventional business, however, in x-contracts they are not acceptable because they are exactly what x-contracts are meant to prevent.

To be consistent with our arguments we show the English text of the example clauses discussed in Section 3.3.2 after editing them to correct the ambiguities that were present:

1 Offer
1.1 The supplier may use his discretion to send offers to the purchaser.
1.2 If no offer is sent within seven days after the signature of the x-contract, or after the latest rejected offer, the x-contract shall be terminated.
1.3 The purchaser is entitled to accept or reject the offer, but he shall notify his decision to the supplier within five days after the receipt of the offer.
2 Commencement and completion
2.1 The contract shall start immediately upon signature.
2.2 The purchaser and the supplier shall terminate the x-contract immediately after reaching a deal for buying an item.

Like in section 3.3.2, from the English text contract clause we will extract the sets of rights and obligations for the purchaser and the supplier and express them in terms of operations for FSMs.

Purchaser's rights:
R_1^P : Accept offers.
R_2^P : Reject offers.
Purchaser's obligations:
O_1^P : Start the x-contract.
O_2^P : Respond within 5 days after receipt of an offer.
O_3^P : Terminate the x-contract.
Supplier's rights:
R_1^s : Send offers within 7 days after start of the x-contract.
Supplier's obligations:
O_1^S : Start the x-contract.
O_2^S : Terminate the x-contract.

Apart from minor changes, the rights and obligations look the same as the ones listed in Section 3.3.2 . For example O_2^P must happen within 5 days, and this will be reflected in the FSMs. An interesting right is R_1^s because it is a right but also there is an obligation to perform it within a time limit.

Once again as in Section 3.3.2 we show how the sets of Rights and Obligations are mapped into FSMs in figure. 3.6.

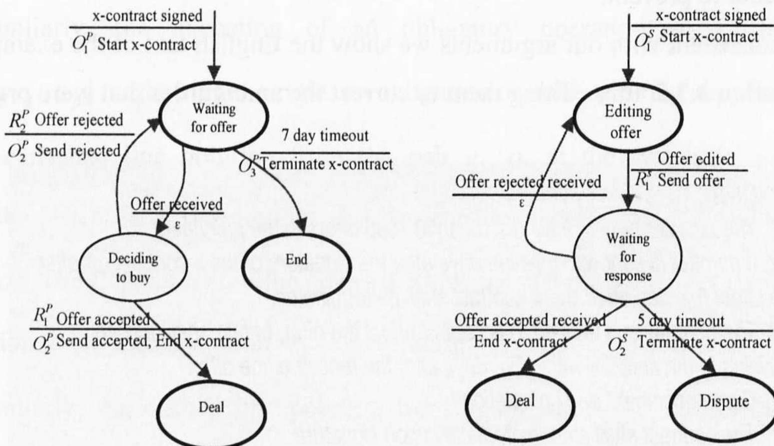


Fig. 3.6. FSM Representation of an unambiguous contract for the purchase of goods

We have to admit that though the FSM model we present for this rather simple x-contract looks correct at first glance, we do not guarantee it is completely free from inconsistencies. We argue that it is too adventurous to claim that the electronic representation of a contract is free of inconsistencies before the model that describes it is validated using formal tools such as the Spin validator (see Chapter 4).

What can be done if the validator discovers that the model of an x-contract contains some inconsistencies? We have found out that this situation is rather common with existing

standard contracts. We strongly argue that the existence of inconsistencies in a standard English text contract is going to be the normal rather than exceptional.

Because of this, we believe that, except for the fairly well standardized contracts (see Section 3.4.3) the conversion of a contract into an x-contract is an interactive process. The interaction involves the lawyer and the technical person in charge for the implementation of the x-contract:

1. The lawyer edits the English text contract.
2. The technical person converts the contract into a formal model and validates it.
3. If inconsistencies are discovered in the contract, the technical person goes back to the lawyer (point 1) to request him to correct the English text, taking care that the main purpose of the contract is not changed.
4. If the validator indicates that the model is free from inconsistencies and the lawyer is satisfied with the last version of the English text contract, the technical person proceeds to convert the model into the actual program that will enact the x-contract.
5. Once the English text contract and the x-contract are ready, the contracting parties can sign and enact it.

In Chapter 5, we present an example of a contract that we have converted from English text into FSMs. It is a contract for the purchase and supply of e-goods and is inspired in an example for the purchase and supply of goods published in [GM00]. We changed the original text to make it more illustrative in terms of essential x-contract interactions. Most importantly, we changed the original text to eliminate several ambiguities that prevented the contract from being described with FSMs.

3.4.3 Ready to fill in, sign and enact x-contracts

The manual conversion process discussed in Section 3.4.2 from the English text contract to the electronic contract is not the best solution.

The ideal scenario would be one where an English text contract can be converted by a lawyer into an x-contract that monitors and enforces an agreement, without the intervention of a technical person. Yet this is currently unrealistic, if this were possible then we would not need programming for any applications. Automation of the contract conversion process with current technology however, can be achieved for standard contracts.

In the business world, there is a family of applications where the contracting parties resort to fairly standardized contract templates which are offered ready to be filled in and signed. Examples of these templates are tenant agreements. These contract templates can be

bought at the stationery. They are offered on the take-it-or-leave-it basis since the clauses of the contract are not negotiable. The contracting parties can negotiate the data to be written in the blanks, but nothing else.

We believe that for this family of applications it is possible to offer (possibly in return for a fee) ready to fill in and sign x-contracts. We can think of a Web place where standard English text templates are stored together with their inconsistencies-free x-contracts. The contracting parties can then remotely fill in the template that suit their requirements, sign it, pay for the service and enact the x-contract.

The steps that would be required to enable this can be summarized as follows:

Defining Standard contracts

1. Lawyers compile and edit a number of standard contracts used frequently as "standard" between certain entities.
2. The technical person converts each of the standard contracts into FSMs and validates them.
3. If inconsistencies are discovered in a standard contract, the technical person reviews the contract with the relevant lawyer, and the English text is corrected.
4. If the validation process indicates that the FSM representing the contract is free from inconsistencies, and the lawyer is satisfied with the standard contract, the standard contract is added to the standard contract data base with its FSM.

Agreement phase

1. The trading partners use a contract editor to access the data base and choose a standard contract relevant to them.
2. They fill in the relevant data (deadlines, prices, etc.)
3. Once satisfied they both electronically sign the contact.

Enactment phase

1. Both parties take the signed contract FSM and "plug it in" the contract monitoring system.
2. The contract monitoring system uses the FSM code (which is the x-contract code) to monitor and enforce the contract agreements between the parties.

3.5 Summary

Before attempting to implement an x-contract electronically; the clauses within the original conventional text contract must precisely abstracted and the parties' rights and obligation must be mapped into computer code convertible mathematical notation, also the ambiguities that exist within the original conventional text contract must be detected and removed.

To specify party interaction related rights and obligations, we propose the use of finite state machines. Thanks to their graphic nature, finite state machines are easy to read. On the other hand the mathematical theory behind them makes them useful for ensuring the correct operation of an x-contract.

In this chapter we described and proposed a method by which contracts' rights and obligations can be mapped into FSMs. The important issue of validation of the correctness of the FSM contract model resulting from the conversion process is discussed in depth in the next chapter.

Blank Page

Chapter Four

Validation of Electronic Contracts

Even for the simplest communication systems, it is difficult to design correct protocols, and even more difficult is the task of validating the correctness of a protocol's procedure rules, i.e. correctness of the logic that describes the interaction between processes.

Because of this, the use of verification languages to write the procedures rules and software tools to verify the correctness of the resulting code -called the validation model- have been widely used for validating the correctness of protocol implementation.

In the previous chapter, we argued that from the point of view of the interaction and synchronisation between the parties involved, x-contracts are equivalent to communication protocols. Therefore, as is the case with communication protocols, designing a correct model of an x-contract that is free from inconsistencies is going to be very difficult. Such a model can be written in finite state machine or other formal notation with means for validating the correctness of communication protocols.

In this chapter, we illustrate how verification systems and verification languages can be utilized to simplify the process of designing x-contracts. By putting into practice existing machinery that was originally developed for validating communication protocols, we hope to exemplify one of the important benefits of employing FSMs for the design of x-contracts.

4.1 A Verification Language - Promela

So far one of the most successful software tools used to trace logical design errors in distributed systems and in particular in communication protocols is *Spin (Simple Promela INterpreter)*. Spin is a generic verification system that accepts design specifications written in the verification modelling language called *Promela (PROcess MEta Language)*. In this section, we will discuss this modelling language and leave the discussion of *Spin* until Section 4.2.

Promela is *Spin*'s input language and provides a vehicle for making abstractions of protocols so that details that are unrelated to the communication processes are suppressed. A *Promela* program consists of processes, message channels and variables. The state of the whole system depends on the state of these three components.

A validation model is a piece of code that describes the procedure rules, i.e. the interaction between processes. Having the code and a simulator to execute it, the verification

of the completeness of the protocol and its logical consistency (free from deadlocks for example) is achievable and furthermore, the implementation of the system follows from converting the *Promela* code to a high-level one, C or C++ for example. The difference between a *Promela* version of the protocol and the final high-level language implementation is that the former deliberately abstracts from issues of protocol design, such as message format, neither does it say how a message is to be transmitted, encoded, decoded, stored, etc. Moreover, it does not deal with details irrelevant to processes' interaction such as encryption and decryption of messages and implementation of timers.

The syntax of *Promela* is described by Holzmann in the appendix C of his book [H91]. However, to help the reader understand our *Promela* code, we introduce the basic *Promela* statements and their semantics here.

executability In *Promela* the execution of a statement is conditional on its executability, i.e., at a given moment of time a statement is either executable or blocked depending on the state of a variable or channel. Executability is the basic means of synchronization; hence, as shown below in *send/receive* statements, input and output through a channel allows the communication between two processes and synchronization as well. For example, the statement "***if (a == b) a = a+1 fi***", either increments the value of *a,* or blocks until the condition "*a == b*", holds.

send The syntax of the send statement is

channel ! var

where **channel** is the name of a channel and **var** is a variable that holds a message.

receive The syntax of the receive statement is

channel ? var

where **channel** is the name of a channel and **var** is a variable that contains a message.

separators → and ; are separators.

skip is a null statement. It is always executable and its execution has no effect. It is normally used to satisfy syntax requirements.

goto The goto statement works as the infamous goto of high level languages, it transfers control to any labelled statement. Like the **skip** statement, **goto** is always executable. As *Promela* pays no attention to the problem of programming techniques it lacks most of the constructs for writing a well-structured code, as a result **goto** is intensively used.

If-fi selection A selection statement begins with **if** and ends with the keyboard **fi** and contains a list of one or more options. Every option begins with the flag :: followed by a Boolean expression called a *guard.* An option can be executed only if its guard is executable. Only one option from the list is executed. If more than one guard is executable, one of them is selected at random and the corresponding option is executed. If all guards are unexecutable, the process blocks until at least one of them becomes executable. In the following example the variable counter is either incremented or decremented depending on the value of a and b

```
if
:: (a== b) → counter = counter +1
:: (a != b) → counter       = counter -1
fi
```

do-od repetition This statements works in a similar way as the **if-fi** one, but it is repeated until a **break** statement is encountered or an unconditional **goto** jump is performed. In the example shown the program loops until either the variable counter is decremented to zero (loop stopped with the break statement) or until for some mysterious reason, the counter decrements to below zero (loop ended using a **goto** jump to a label we give the name "Error" for example).

```
do
:: (counter < 0) → goto Error
:: (counter==0) → break
:: (counter > 0) → counter = counter - 1
od
```

timeout This statement represents a condition that eventually becomes true if and only if no other statement in the block of commands is executable.

4.2 A Verification System - Spin

Spin is a generic validation system that supports the design and verification of asynchronous process systems [H97]. *Spin* verification models are focused on proving the correctness of *process* interactions, and they attempt to abstract as much as possible from internal sequential computations. It was developed at Bell Labs in 1980, its source code written in ANSI standard C, and can be easily downloaded from the Internet [SP03] and compiled for UNIX, Linux, and Windows platforms.

Spin accepts design specifications written in the verification language *Promela* (see Section 4.1), and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL) [LTL].

The *Spin* package consists of two independent tools a "Simulator" and a "Validator" that are meant to be used at different stages of the protocol validation process.

Spin and its commands, can be executed directly from the command line, however it is probably more useful to use the features of the graphical user interface *XSpin*. The "Basic Spin Manual" [H97], is a very useful document about how to run *Spin* from the command line.

4.3 XSpin

The easiest way to get started with *Spin* is to run the graphical interface *Xspin*, see figure 4.1. The graphical interface runs independently from *Spin* itself, and helps to invoke the proper *Spin* commands based on menu selections. *Xspin* runs *Spin* in the background to obtain the desired output, and wherever possible it will attempt to generate a graphical representation of such output. *Xspin* knows when and how to compile promela code for the model checkers that *Spin* can generate, and it knows when and how to execute it, so there is less commands that the model designer needs to remember.

More details and tutorials on running *Spin* using *XSpin* can be found at [SP03].
The *Help* menu option (see figure 4.1), provides very good explanations of the many capabilities of *XSpin*. Also our examples in this chapter and next will be described in sufficient detail, so that the reader gets a good idea of *XSpin's* functionality.

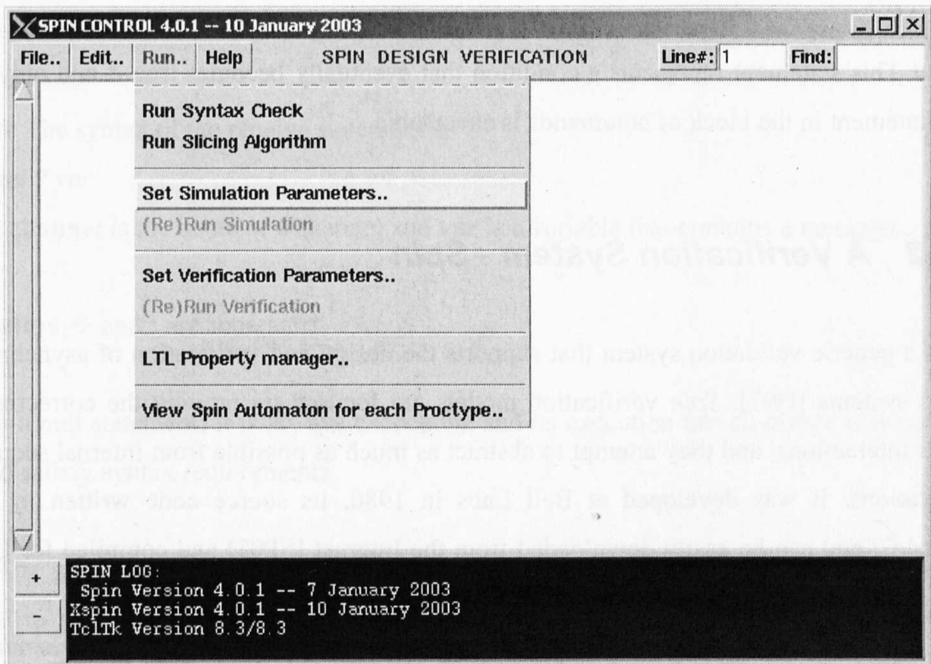


Fig.4.1. The Graphical user interface XSpin

4.4 The Spin Simulator

As it names implies, the simulator can simulate the execution of a validation program (a model in *Spin* jargon) written in *Promela*. It simulates *Promela* code by interpreting its statements on-the-fly. To do its job the simulator performs a single-pass verification procedure making effort to save memory and CPU resources; it tries to store in memory just enough information to complete the verification process and to verify the correctness of the requirements but for the smallest possible fragment of the whole behaviour of the system. For example, if at a given point during the simulation process the simulator is faced with more than one executable statement (a nondeterministic choice), it selects just one. This means the simulator does not perform any exhaustive reachability analysis but goes only through a single sequence of reachable states in the system, which is chosen depending on the value of the seed the random number generator is initialized with, if no seed value is specified, the simulator chooses one randomly [H91].

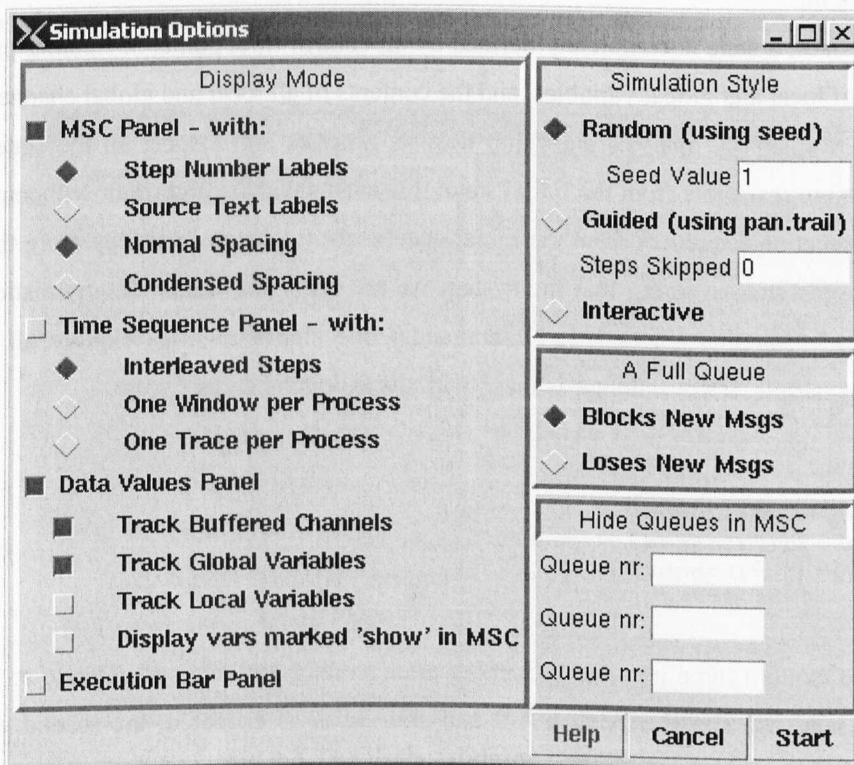


Fig.4.2. The XSpin simulator interface

The advantage of using the simulator at an early stage of the system design is that it can immediately tell the system designer about simple inconsistencies in his protocol, such as deadlocks, and unspecified receptions. It is fast and does not demand a great deal of computer resources since it does not need to construct a global state for the system. Because of this,

systems of arbitrary size can be easily simulated. However, since it runs a random simulation only, the absence of errors reported by the simulator does not necessarily mean that the system is error-free. The accurate verification of a system is performed by the *Spin* validator.

The simulator is simply executed using *Xspin,* by selecting the *RUN* menu option, and then *Set Simulation Parameters,* where the designer can specify different options to inform the simulator to output to the screen; what messages are sent or received and by which processes, what line of the code is executed, the value of local and global variables, the value of the seed for the random number generator, and so on. See figure 4.2.

4.5 The Spin Validator

The job of the *Spin* validator is to validate the correctness requirements *(also called correctness criteria and properties)* of *Promela* code given at its input.

Spin belongs to the category of protocol verification systems that are based on the analysis of the reachability of system states. Before going further in our discussion let us define what a state is in *Spin.*

In *Spin,* a state is completely defined by all control flow points of running processes, all values of local and global variables, and the contents of all local and global channels.

A reachability analysis algorithm tries to generate and inspect all the states of the system that are reachable from the initial state; this means that the algorithm will construct all possible execution sequences from the initial state to the final state (possibly more than one). In other words and assuming that the system we are analysing is non-deterministic (i.e. its Promela code contains guarded "::" commands), the algorithm must explore all possible paths. For example, if the validator is faced with the following code:

```
...
if
:: (a > 0) → statement1
:: (a = 0) → statement2
:: (a < 0) → statement3
fi
...
```

Spin has to explore three possible sequences after reading the value of variable a: The first execution sequence considers that a > 0 and *statement1* is executed; the second execution sequence considers that a = 0 and *statememt2* is executed; for the third execution sequence, the validator assumes a < 0 and executes *statement3.*

It is worth noting that for a validation to be possible, the *Promela* specification of the system must restrict the number of processes, flow control point, variables, channels, and slots of channels to a finite number so that the number of states of the system remains finite and the system can be analysed exhaustively by enumerating its reachable states.

Depending on the size of the system, the generation and analysis of all possible states can be computationally unfeasible. Most of the time the designer of a large system (more than 10^5 reachable system states) is faced with the state space explosion problem. To understand this, we will briefly discuss how Spin works.

A system is represented in a *Promela* model as a set of processes. Spin translates each process into a finite state automaton. Next, the asynchronous interleaving product of automata is computed and translated into an automaton. This automaton represents the global system behaviour and is called the *state space of the system* or *the global reachability graph*.

A correctness requirement of a system is expressed in a formal notation called *Linear Temporal Logic* (LTL for short). LTL can be translated into what is known as the *Buchi automaton*.

To perform verification, *Spin* computes the synchronous product of the *Buchi* automaton and the automaton that represents the global system behaviour. The result of this computation is another *Buchi* automaton and is used by *Spin* to see what language it accepts. If such a language is empty, this means that the correctness requirements expressed in the LTL formula are not satisfied by the system.

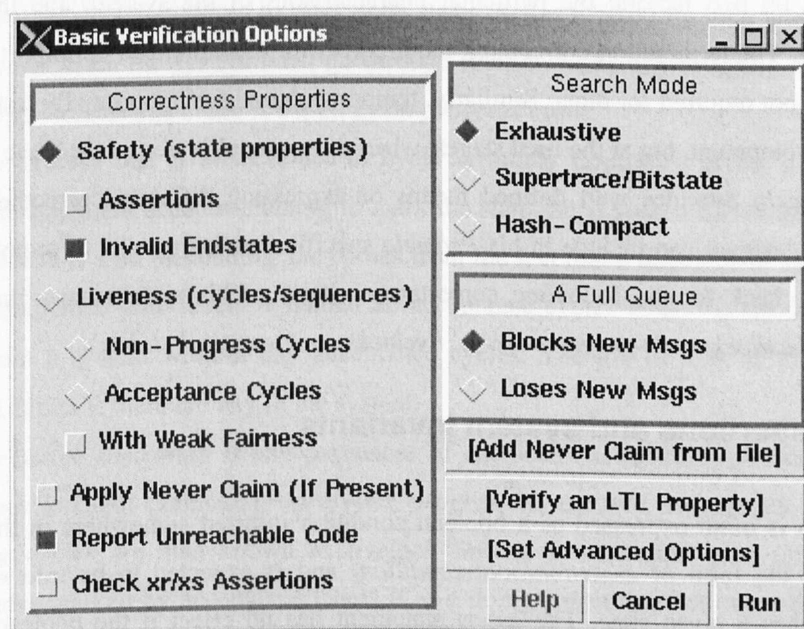


Fig.4.3 The Spin validator interface

Something to keep in mind during validation is to tell whether the language accepted by the *Buchi* automaton is empty or not. *Spin* has to generate and verify all possible sequences of states of the automaton; this can become prohibitively expensive since in the worse case, the state space of the system has the size of the Cartesian product of all its components: control flow points, processes, local and global variables, and channels.

Once the system is written in *Promela* code and passed through the simulator, the designer is encouraged to validate it using *XSpin.*

The *Basic Verification Options* menu in figure 4.3, presents the user with a number of options that comprise the most common correctness requirements that the user might need to perform his verification. Correctness requirements are the subject of the next section.

4.6 Correctness Requirements

A crucial decision the designer of a protocol or a *FSM* model of an x-contract, has to make, is what correctness requirements (absence of deadlocks, mutual exclusion, temporal claims, etc.) to check his system on. This is extremely important not only because this will guarantee that the system is free of a particular kind of error, but also because the inclusion or exclusion of one of these requirements can have significant impact on the number of total states of the system and for instance on RAM memory and CPU time demand to validate them.

Although the correctness requirements that are usually validated in protocols are well-known, the list of correctness requirements the protocol designer tests his/her protocol on, depends on two factors: the particular characteristics of the system, and the stage of development. The termination correctness requirement for example, can be important for one protocol but not required for other. Similarly, temporal claims are not normally tested in early stages of development, but at the final stages, when the protocol is free of the basic errors.

Promela provides well defined means of expressing different correctness criteria: namely, the designer can include in his *Promela* specification statements to prompt the *Spin* validator to check for the following correctness criteria of his system: *assertions, system invariants, deadlocks, non-progress cycles, livelocks, and temporal claims.*

4.6.1 Assertions and system invariants

An assertion is often expressed as a boolean condition inserted somewhere in the *Promela* code. It has the form of *assertion(bool-condition)* and is expected to be true whenever a process reaches a given state. The *assert* statement has no effect if the boolean condition holds true; conversely, it generates an error message if the boolean condition becomes false.

If the designer wants a boolean condition to remain true in all reachable system states he/she can express this as a system invariant. A system invariant is just a generalization of an assertion, it has the same form, *assertion(bool-condition)*, and is placed in a separate process that runs concurrently with the one the designer wants to validate; the assert statement is executed precisely once for every state of the system.

4.6.2 Deadlocks

Since *Spin* expects only systems with a finite number of states, it expects that the system under validation either terminates after a finite number of state transitions or it goes back to a previously visited state (a loop). Both alternatives are considered a valid end to a process. Although the second alternative is not the final state of the system, it is considered and called a proper end state in *Spin*. If the system does not match this correctness criterion it is said to have a deadlock. In *Promela*, a proper end state is identified by a three-character prefix *end-state* label which has the form of end*something*, where *something* is any sequence of characters accepted by *Promela* in names used as identifiers. Example of *end-state* labels are: *end, endcycle, end0, end1*, and so on.

4.6.3 Progress cycles and livelocks

In *Promela* (and other programming languages) infinite cycles are considered correct behaviour for a process as long as the process goes through the states the designer expects. To express that a process cannot cycle infinitely without visiting certain states, *Promela* provides the statement progress*something* to mark such states. States marked by such labels are called *progress-states* since the system must go through them to make any progress. An execution sequence that violates this claim is called a *non-progress cycle*.

To express that it is incorrect to cycle infinitely through a given state, *Promela* provides the statement accept*something* to mark the state. Such state is called an *acceptance-state*. The name is a bit misleading and comes from the fact that a sequence of statements that contains acceptance-state labels is named an acceptance cycle. What we are saying here is that we want a system without any acceptance cycles. The job of *Spin* is to detect these acceptance cycles if there are any in the system.

As before *something* is any sequences of characters accepted by Promela in names used as identifiers. For example, *progress-svr, progressClt, accept0, accept1*, etc.
Acceptance cycles are also known as *livelock* since a process that goes infinitely often through states marked by acceptance labels is still doing something but trapped in a loop. It cannot escape from there and go through the states the designer wants it to go through.

4.6.4 Temporal claims

In some cases it is necessary to express that a state in which a certain condition is true cannot be followed by a state in which that condition or a different one is false. For example, the designer might want to express that if it is true that a channel with a single slot is full, it

cannot remain full after reading a message from it. In *Spin* these correctness requirements are called temporal claims, and in *Promela* are expressed with the help of the statement:

never{Prom_statement1, Prom_statement2, Prom_statement3,...}

Each *Prom* statement is a *Promela* statement that contains the details of the claim; for example; assertions, progress-states, and acceptance-states labels.

4.6.5 Safety and liveness properties

In protocol validation, properties are grouped into two major classes: safety properties and liveness properties. Informally, a safety property states that nothing bad ever happens. Let us take a lift as an example. A safety property will state that if the lift is travelling or stopped between two levels its door will never open. On the other hand, a simple liveness property states that something good will eventually happen. Again, let us take a lift as an example. A simple liveness property will state that if a user has arrived at the intended floor, the doors will eventually open. In other words, the passenger will eventually terminate his journey.

Another way of explaining safety and liveness properties is by saying that a safety property states what we do not want the system to do. Conversely, a liveness property states what we want the system to do.

These two concepts have been widely used in the literature devoted to correctness of concurrent programs since they were introduced by Lamport [L77].

In *Spin* the concept of safety properties is used to group together assertions and system invariants, deadlocks, and unspecified receptions. On the other hand, non-progress cycles, livelocks, and temporal claims fall in the class of liveness properties.

As explained in Section 4.6, the designer can use *Spin* directives to instruct the validator to validate the properties he is interested in. It is a well-known fact that it is always simpler specifying what we do not want from a system than specifying what we want, thus, it makes sense to begin the validation of a protocol by validating safety properties first and leave liveness properties for the last stages of the validation.

The reader interested in more details about safety and liveness properties is encouraged to refer to [H91] where these concepts are studied in depth.

4.6.6 Cost of correctness requirements

We have just discussed what correctness criteria can be specified in Promela to be validated by *Spin*, the order in which we introduced them reflects the level of sophistication in the validation and at the same time the cost of performing the validation in terms of RAM memory and CPU time demands.

Holzmann reports [H91] that it is comparatively cheap to validate assertions and absence of deadlocks. The computational cost for this is linear in the number of reachable states *(R)* of the system both in RAM memory space and CPU time. To check on progress cycles and livelocks can be twice as hard in terms of CPU time but there is not a noticeable increase in RAM memory requirements. The most expensive correctness criterion to validate is temporal claims. Compared to assertions and absence of deadlocks validation, the cost can be *2N* times as hard, where *N* is the number of reachable states in the sequence of statements contained in the claim:

never{Prom_statement1, Prom_statement2, Prom_statement3,...}

It is important and helpful to notice that *Spin/XSpin* allow us to validate these correctness criteria separately (see Section 4.6), for example check the system for non-progress cycles only, or for acceptance cycles only so that the simpler requirements do not contribute to the cost of the more sophisticated ones.

The selection of the correctness criteria to validate in each run is made with the help of *XSpin* options: For example, the designer can select the *Safety (state properties)* option, to indicate that he/she is interested only in validating safety properties. The definition of safety is explained in Section 4.6.5. Similarly, checking the option *Non-Progress Cycles* indicates that the designer wants to check on non-progress cycles (figure 4.3)...etc.

Once the required correctness requirements are selected, the user can simply select the *Run* option to begin the validation.

4.7 Basic Verification of x-contracts

In the first three sections of this chapter, we presented the reader with a brief introduction to the verification language *Promela*, the verification system *Spin*, and to the requirements that the verification system will need to check for correctness in order to verify the accuracy and correctness of a protocol. But how do we make use of such verification tools for specifically verifying the correctness of a *FSM* representation of an x-contract? To answer this, we shall reintroduce the example presented in Chapter 3. The example was a contract for the supply of e-goods between a *Supplier* and a *Purchaser*.

Firstly in Section 4.7.1, we will present the contract in its initial ambiguous state, extract from its clauses the sets of rights and obligations of the contracting parties, and map them into finite state machines. We will then attempt to verify the correctness of the FSMs by creating a verification model using *Promela*, and then using *Spin* to verify the correctness of the model against some correctness requirements. In Section 4.7.2, we will present the

rewritten contract after taking care to eliminate any ambiguities that we discovered (if any) using *Spin*. And as in Section 4.7.3 we will extract the rights and obligations, map them into FSMs, create the verification model, and validate its correctness against the same correctness requirements.

4.7.1 Contract before removal of ambiguities

This deed of agreement is entered into as of the effective date identified below.
Between
[Name] of [Address] (To be known as the (Supplier)), and [Name] of [Address] (To be knows as the (Purchaser)).
Whereas
(Supplier) desires to enter into an agreement to supply (Purchaser) with [Item].
Now it is hereby agreed that (Supplier) and (Purchaser) shall enter into an agreement subject to the following terms and conditions:
1. Definitions and Interpretations
1.1 Price, Dollars or $ is a reference to the currency of the [Country].
1.2 All information (purchase order, payment, notifications, etc.), is to be sent electronically.
1.3 This agreement is governed by [Country] law and the parties hereby agree to submit to the jurisdiction of the Courts of the [Country] with respect to this agreement.
2. Offer
2.1 The supplier may use his discretion to send offers to the purchaser.
2.3 The purchaser is entitled to accept or reject the offer, but he shall notify his decision to the supplier.
3. Commencement and completion
3.1 The contract shall start immediately upon signature.
3.2 The purchaser and the supplier shall terminate the x-contract immediately after reaching a deal for buying an item.
4. Disputes
4.1 (Supplier) and (Purchaser) ***shall*** *attempt to settle all disputes, claims or controversies arising under or in connection with the agreement through consultation and negotiations in good faith and a spirit of mutual cooperation.*
4.2(Supplier) and (Purchaser) ***shall*** *provide electronic evidences about breaches of the e-contract.*
4.3 This method of determination of any dispute is without prejudice to the right of any party to have the matter judicially determined by a [Country] Court of competent jurisdiction.
5 Amendment
5.1 This agreement ***may*** *only be amended in writing signed by or on behalf of both parties.*
E-SIGNATURES
In witness whereof (Supplier) and (Purchaser) have caused this agreement to be entered into by their duly authorized representatives as of the effective date written below.
Effective date of this agreement: [day] of [month] [year]

[E-signature]	*[E-signature]*
[Person]	*[Person]*
[Role]	*[Role]*

E-address for Notices:

[E-address]	*[E-address]*

Fig.4.4. Contract for the purchase of goods between a purchaser and a supplier

The example in figure 4.4 is a contract signed between a *Purchaser*, and a *Supplier*. The contract in its original state, before removing any ambiguities can be abstracted by the following clauses:

1 Offer
1.1 The supplier may use his discretion to send offers to the purchaser.
1.2 The purchaser is entitled to accept or reject the offer, but he shall notify his decision to the supplier.
2 Commencement and completion
2.1 The contract shall start immediately upon signature.
2.2 The purchaser and the supplier shall terminate the x-contract immediately after reaching a deal for buying an item.

The rest of the clauses in the contract (figure 4.4) have been left out because we are only concerned with the clauses required for controlling and monitoring the interactions between the parties.

Rights and Obligations:

From this English text contract we can extract the sets of rights and obligations for the *Purchaser* and the *Supplier*.

Purchaser's rights:
R_1^P : Accept offers.
R_2^P : Reject offers.
Purchaser's obligations:
O_1^P : Start the x-contract.
O_2^P : Reply to offers.
O_3^P : Terminate the x-contract.
Supplier's rights:
R_1^S : Send offers.
Supplier's obligations:
O_1^S : Start the x-contract.
O_2^S : Terminate the x-contract.

After extracting the sets of rights and obligations, we can represent them in *Finite State Machines* as shown in figure 4.5.

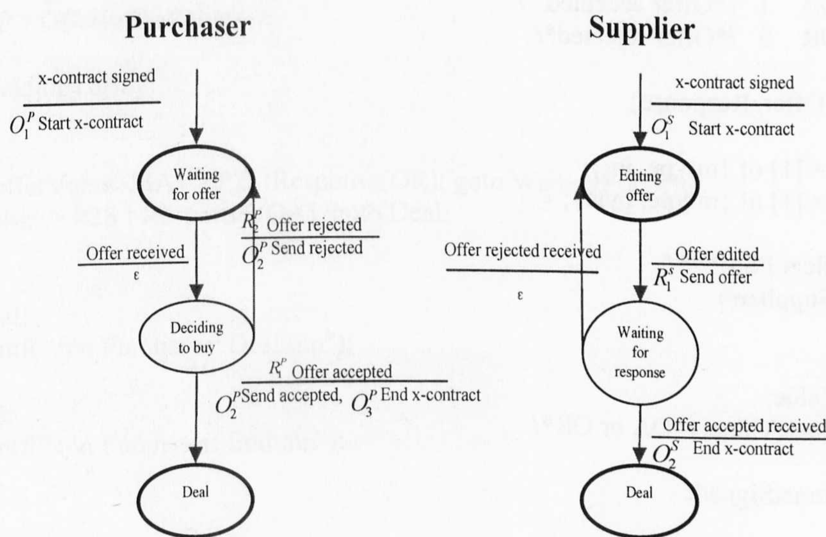


Fig.4.5 FSM Representation of an ambiguous contract for the purchase of goods.

Our next task is to represent the FSM in figure 4.5 in the modeling language *–Promela-* that is the input language of the verifier *Spin*. The complete verification model is presented next:

```
/*Verification Model for the Contract Finite State Machines*/
/*in their initial ambiguous state*/

#define                MA   20 /*Maximum acceptable offer*/
#define  OA    1  /*Offer accepted */
#define  OR    0  /*Offer rejected*/

mtype = {Offer, Response}
chan S2P = [1] of {mtype, int};
chan P2S = [1] of {mtype, byte};

proctype Supplier() /***Suppliers FSM***/
{
 int offerValue;
 byte responseValue; /*OA or OR*/
  SupEContractSigned:
  EditingOffer:
if
   :: offerValue = 30; /* An offer that is too high > MA*/
   :: offerValue = 20; /* < MA */
   :: offerValue = 10; /* < MA */
 fi;
 if
   :: S2P!Offer(offerValue) -> goto WaitingForResults;
   :: skip /*Taking into account the possiblity that*/
 fi;      /*the supplier might not send anything */
  WaitingForResults:
 P2S ? Response(responseValue);
  if
   :: (responseValue == OR) -> goto EditingOffer;
   :: (responseValue == OA) -> goto Deal;
fi;
 Deal:
 printf("\n\n Supplier: Deal \n\n");
 end:
 printf("\n\n Supplier: End \n\n");
}

proctype Purchaser() /***Purchasers FSM***/
{
 int  offerValue;
 PurEContractSigned:
 WaitingForOffer:
 S2P ? Offer(offerValue) ->
 DecidingToBuy:
 if
   ::(offerValue>MA)-> P2S!Response(OR);
        goto WaitingForOffer;
   :: else -> P2S ! Response (OA); goto Deal;
 fi;
 Deal:
 printf("\n\n Purchaser: Deal\n\n");
 end:
 printf("\n\n Purchaser: End\n\n");
}

init
{
 run Supplier();
 run Purchaser();
}
```

```
1.  /*Verification Model for the Contract Finite State Machines*/
2.  /*in their initial ambiguous state*/
3.
4.  #define        MA   20 /*Maximum acceptable offer*/
5.  #define  OA    1  /*Offer accepted */
6.  #define  OR    0  /*Offer rejected*/
7.
8.  mtype = {Offer, Response}
9.
10. chan S2P = [1] of {mtype, int};
11. chan P2S = [1] of {mtype, byte};
12.
13. /***Suppliers FSM***/
14. proctype Supplier()
15. {
16.
17.  int offerValue;
18.  byte responseValue; /*OA or OR*/
19.
20.  SupEContractSigned:
21.
22.  EditingOffer: /*under this label/state, we must take into account*/
23.             /*All possible actions that the supplier coould take*/
24.  if
25.  :: offerValue = 30; /* An offer that is too high > MA*/
```

```
26. :: offerValue = 20; /* < MA */
27. :: offerValue = 10; /* < MA */
28. fi;
29.
30.
31. if
32. :: S2P!Offer(offerValue) -> goto WaitingForResults;
33. :: skip /*Taking into account the possiblity that*/
34. fi;    /*the supplier might not send anything */
35.
36. WaitingForResults:
37. P2S ? Response(responseValue);
38.
39. if
40. :: (responseValue == OR) -> goto EditingOffer;
41. :: (responseValue == OA) -> goto Deal;
42. /*:: else -> printf("\n\n Error\n\n");*/
43. fi;
44.
45. Deal:
46. printf("\n\n Supplier: Deal \n\n");
47.
48. end:
49. printf("\n\n Supplier: End \n\n");
50.
51. }
52.
53. /***Purchasers FSM***/
54. proctype Purchaser()
55. {
56. int  offerValue;
57. PurEContractSigned:
58.
59. WaitingForOffer:
60. S2P ? Offer(offerValue) ->
61.
62. DecidingToBuy:
63.
64. if
65. ::(offerValue>MA)-> P2S!Response(OR); goto WaitingForOffer;
66. :: else -> P2S ! Response (OA); goto Deal;
67. fi;
68.
69. Deal:
70. printf("\n\n Purchaser: Deal\n\n");
71.
72. end:
73. printf("\n\n Purchaser: End\n\n");
74. }
75.
76. init
77. {
78. run Supplier();
79. run Purchaser();
80. }
```

To clarify the conversion process from an *FSM* to its *Promela* verification model, we suggest that every state in the *FSM* has an address label in the verification model.

In this code, there are two asynchronously executing processes, a *Purchaser*, and a *Supplier*. There are two message types (*mtype*) exchanged between the two processes, the "*Offer*" made by the *Supplier*, and the "*Response*" to the *Offer* made by the *Purchaser*. In order to exchange these messages, we have two message channels, *S2P* (*Supplier* to *Purchaser*), and *P2S* (*Purchaser* to *Supplier*).

Note that we make the assumption that acceptance or rejection of an offer is based only on the price offered.

The process begins with the *Purchaser* waiting for an *Offer* message with an *offerValue* from the *Supplier*. Under the address label *EditingOffer*, the *Supplier's offerValue* is chosen randomly in a "*if ... fi;*" structure. The *offerValue* is then either sent or not sent (randomly) through the *S2P* channel. The *Supplier* now waits for a response to his/her offer, while the *Purchaser* makes a decision on whether to accept it or reject it based simply on the price offered. If the price is satisfactory, the *Purchaser* sends an *OA* (Offer Accepted) message to the *Supplier*, the *Supplier* receives this, and they both go into the end "*Deal*" state. If however, the price offered by the *Supplier* is too high (greater than *MA*), then the *Purchaser* sends an *OR* (Offer Rejected) message, and they both go back to the beginning of the process. The *Supplier* can then make another offer, and so on.

Simulation Run

The "*Run Syntax Check*" option from the "*Run*" menu, in the *Spin* validator, checks the model for syntax errors. After doing this, we can use *Spin* for one or more simulation runs before running the validator. As we point out in Section 4.4, the simulator performs a single pass verification procedure. The advantage of using the simulator at an early stage of our design is that it can immediately tell us about simple inconsistencies in our design, such as deadlocks, and unspecified receptions. The simulator is useful also as it provides us with the opportunity to simulate a number of runs before implementation of the design.

Figure 4.6, shows the sequence chart of a random simulation run. And figure 4.7 shows the detailed simulation output. The simulator will repeat a random run precisely if the seed value for the random number generator is kept the same (see figure 4.2). In this case the seed value is "1". The chart in figure 4.6 shows messages being passed between the Purchaser and the Supplier in a single simulation run.

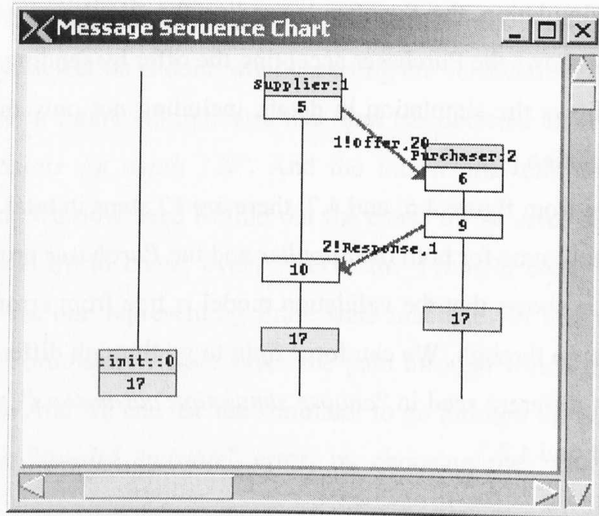


Fig.4.6 Message sequence chart

Three processes are created during the simulation; the *Supplier* process, the *Purchaser* process, and the *init* process. The *init* process, instantiates the *Supplier* and *Purchaser* processes.

```
Simulation Output
  0:    proc - (:root:) creates proc  0 (:init:)
  1:    proc  0 (:init:) creates proc  1 (Supplier)
  1:    proc  0 (:init:) line  78 "pan_in" (state 1)    [(run Supplier())]
  2:    proc  1 (Supplier) line  25 "pan_in" (state 4)  [offerValue = 20]
  3:    proc  0 (:init:) creates proc  2 (Purchaser)
  3:    proc  0 (:init:) line  79 "pan_in" (state 2)    [(run Purchaser())]
  4:    proc  1 (Supplier) line  31 "pan_in" (state 5)  [.(goto)]
  5:    proc  1 (Supplier) line  32 "pan_in" (state -)  [values: 1!Offer,20]
  5:    proc  1 (Supplier) line  31 "pan_in" (state 9)  [S2P!Offer,offerValue]
  6:    proc  2 (Purchaser) line  60 "pan_in" (state -) [values: 1?Offer,20]
  6:    proc  2 (Purchaser) line  60 "pan_in" (state 1) [S2P?Offer,offerValue]
  7:    proc  2 (Purchaser) line  64 "pan_in" (state 8) [else]
  8:    proc  1 (Supplier) line  32 "pan_in" (state 7)  [goto WaitingForResults]
  9:    proc  2 (Purchaser) line  66 "pan_in" (state -) [values: 2!Response,1]
  9:    proc  2 (Purchaser) line  66 "pan_in" (state 6) [P2S!Response,1]
 10:    proc  1 (Supplier) line  37 "pan_in" (state -)  [values: 2?Response,1]
 10:    proc  1 (Supplier) line  37 "pan_in" (state 11) [P2S?Response,responseValue]
 11:    proc  2 (Purchaser) line  66 "pan_in" (state 7) [goto Deal]
 12:    proc  1 (Supplier) line  39 "pan_in" (state 16) [((responseValue==1))]
 13:    proc  1 (Supplier) line  41 "pan_in" (state 15) [goto Deal]
 Supplier: Deal
 14:    proc  1 (Supplier) line  46 "pan_in" (state 18) [printf('\\n\\n Supplier: Deal \\n\\n')]
 Purchaser: Deal
 15:    proc  2 (Purchaser) line  70 "pan_in" (state 10)        [printf('\\n\\n Purchaser:
Deal\\n\\n')]
 Supplier: End
 16:    proc  1 (Supplier) line  49 "pan_in" (state 19) [printf('\\n\\n Supplier: End \\n\\n')]
 Purchaser: End
 17:    proc  2 (Purchaser) line  73 "pan_in" (state 11)        [printf('\\n\\n Purchaser: End\\n\\n')]
 17:    proc  2 (Purchaser) terminates
 17:    proc  1 (Supplier) terminates
 17:    proc  0 (:init:) terminates
3 processes created
Single Step   Suspend                         Save in: sim.out   Clear   Cancel
```

Figure 4.7. Simulation output

The number that is shown in the square boxes is a simulation step number that matches the numbers in the left margin of the *Simulation Output* panel (See figure 4.7). The arrows show the messages being passed between the *Supplier* and the *Purchaser*. So the top arrow shows

that for this random simulation, the *Supplier* is sending the *Purchaser* a price offer of "20". And the bottom arrow shows the Purchaser accepting the offer by sending "1", i.e. "True".

Figure 4.7, shows the simulation in detail, including not only messages passed, but every step of the simulation process.

As we can see from figure 4.6, and 4.7, there are 17 steps in total. And the simulation run ends with the "Deal" state for both the *Supplier* and the *Purchaser* processes.

The simulation shows that the validation model is free from errors for the routes that the simulator chose to go through. We can force *Spin* to go through different simulation paths by simply choosing a different seed in "*choose simulation parameters*", under the *Run* menu option, see figures 4.1 and 4.2.

Validation:

After one or more simulation runs, the next step is to use the Spin validator to validate the correctness of our model against some correctness requirements. We will perform an exhaustive verification run to prove some basic properties, such as absence of deadlocks, unreachable code and states, unspecified receptions, etc. Results of this verification run are presented next in figure 4.8.

```
Verification Output
pan: invalid endstate (at depth 11)
pan: wrote pan_in.trail
(Spin Version 4.0.1 -- 7 January 2003)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (not selected)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid endstates       +

State-vector 44 byte, depth reached 23, errors: 1
      25 states, stored
       1 states, matched
      26 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

2.622   memory usage (Mbyte)

0.00user 0.01system 0:00.04elapsed 23%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (116major+608minor)pagefaults 0swaps
Save in:            Clear    Close
```

Fig.4.8. Verification Output

The section "*Full statespace search for*" illustrates the correctness requirements against which the verification process was performed. A correctness requirement will be followed by

a "+" sign or a "-" sign, indicating whether the correctness requirement was, selected, or not selected respectively. Selection is done when inserting the verification option, see figure 4.3.

The first line in figure 4.8 indicates that *Spin* has detected an error in our verification model. "*invalid endstate (at depth 11)*". And the fourth line tells us that the verification process was stopped. We now need to find out the cause of the error detected. This can be a tedious process if we are to check every line of the *Promela* code, and especially if the verification model was one representing finite state machines of many states, and/or much greater complexity. Fortunately XSpin saves the path through which the verifier detects the error. See Figure 4.9. And we can use the simulator to go through the path through which the verifier detected the "*invalid endstate*" error, by choosing the "*Run Guided Simulation*" option.

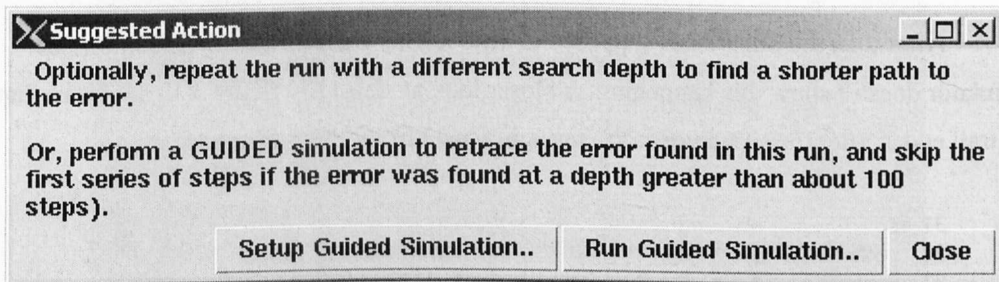


Fig 4.9 Suggested actions for detected error

Figure 4.10, shows the Simulation output of the path through which the error was detected.

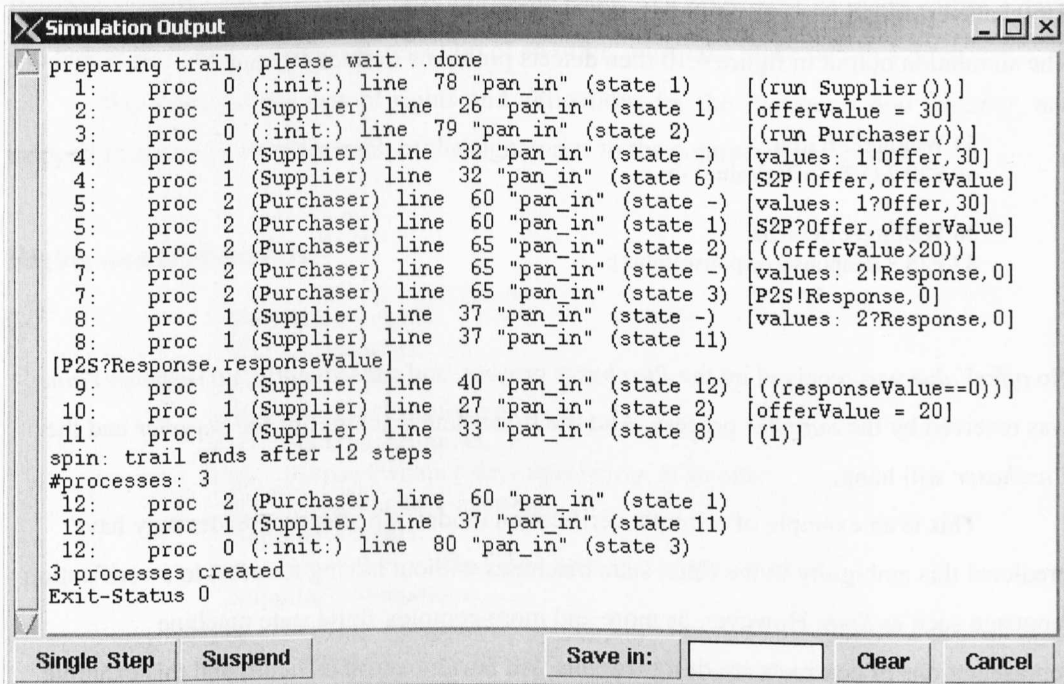


Fig.4.10 Simulation output of erroneous path

Bearing in mind that in *Promela* syntax, "!" means "send", and "?" means "receive". Figure 4.10 shows us that the simulator goes through the following steps:

1. Process init runs the Supplier process.
2. Supplier chooses to offer the Purchaser a price = 30.
3. Process init runs the Purchaser process.
4. Supplier sends the offer value of 30 to the Purchaser.
5. Purchaser receives the offered value.
6. Purchaser detects that the value of 30 is greater than 20.
7. Purchaser sends the response value "0" False = rejected.
8. Supplier receives the response.
9. Supplier detects that the response is a reject.
10. Supplier chooses to offer the Purchaser a price = 20.

After this, the Supplier is supposed to send his/her offer to the Purchaser, but the Simulator doesn't show this happening. A closer look at step 11 in figure 4.10 shows us that the trail ended after the simulator went through line 33 of the *Promela* code:

```
31 if
32 :: S2P!Offer(offerValue) -> goto WaitingForResults;
33 :: skip /*Accounting for the possibility that*/
34 fi;     /*the supplier might not send anything */
```

This line was inserted to take into account that the *Supplier* might choose not to send the offer for whatever reason.

The simulation output in figure 4.10 then detects problems in lines 60, and 37:

```
59 WaitingForOffer:
60 S2P ? Offer(offerValue) ->

36 WaitingForResults:
37 P2S ? Response(responseValue);
```

No *offerValue* was received by the *Purchaser* process, and subsequently, no *responseValue* was received by the *Supplier* process. And the finite state machines of the *Supplier* and the *Purchaser* will hang.

This is an example of a simple verification model. An attentive reader may have predicted this ambiguity in the finite state machines without having to resort to a verification language such as *Spin*. However, as more and more complex finite state machine representations of contracts are designed, this will become more difficult, and this example illustrates the benefits of resorting to such verification languages.

Now that we have detected the problem we will attempt to remedy it next.

4.7.2 Contract after removal of ambiguities

The contract in its initial state is ambiguous because it does not give the parties to the contract, time constraints within which to perform some operations, such as sending messages. This could lead to undesirable situations where one of the parties is waiting indefinitely for a message to arrive. Therefore, the contract after removing the detected ambiguities has the following clauses:

1 Offer
1.1 The supplier may use his discretion to send offers to the purchaser.
1.2 If no offer is sent within seven days after the signature of the x-contract, or after the latest rejected offer, the x-contract shall be terminated.
1.3 The purchaser is entitled to accept or reject the offer, but he shall notify his decision to the supplier within five days after the receipt of the offer.
2 Commencement and completion
2.1 The contract shall start immediately upon signature.
2.2 The purchaser and the supplier shall terminate the x-contract immediately after reaching a deal for buying an item.

As the reader will notice, we have inserted time limits in the form of days, in which the *Purchaser* and the *Supplier* must carry out some tasks. An offer must be made within 7 days, and notification of acceptance or rejection of an offer, must take place within 5 days.

We extracted the sets of rights and obligations for the *Purchaser* and *Supplier*, and mapped them into a *FSM* for each of the signatories to the contract, see figure 4.11.

Rights and Obligations:

Purchaser's rights:
R_1^P : Accept offers.
R_2^P : Reject offers.
Purchaser's obligations:
O_1^P : Start the x-contract.
O_2^P : Respond within 5 days after receipt of an offer.
O_3^P : Terminate the x-contract.
Supplier's rights:
R_1^S : Send offers within 7 days after start of the x-contract.
Supplier's obligations:
O_1^S : Start the x-contract.
O_2^S : Terminate the x-contract.

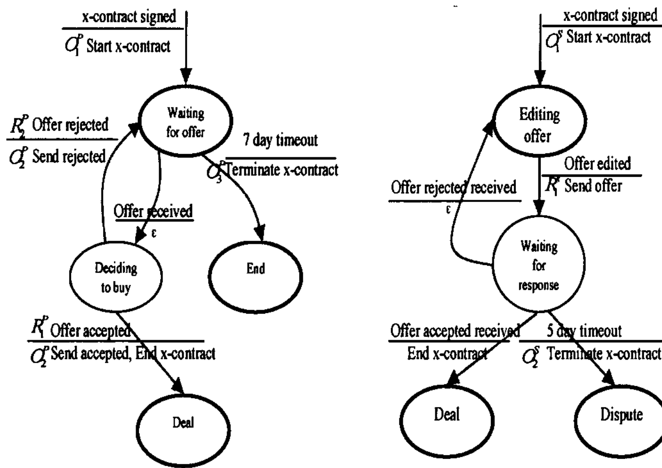


Fig. 4.11. FSM Representation of an unambiguous x-contract for the purchase of goods

In order to check the *FSMs* in figure 4.11 for correctness, we will convert them into the verification language *Promela.* The above *FSMs* are basically modifications of the *FSMs* of the ambiguous contract of Section 4.7.1. To translate these modifications into *Promela,* we will make use of the *Promela "timeout"* statement.

This statement allows a process to abort waiting for a condition that can no longer become true, for example an input from an empty channel. We will add clarification to the code, by illustrating which states in the *FSMs* are reflected in *Promela.* We next present the *Promela* code for the corrected FSM's in figure 4.11:

```
1.  *Verification Model for the Finite State Machines*/
2.  /*after making corrections and removal of ambiguities*/
3.
4.  #define      MA   20 /*Maximum acceptable offer*/
5.  #define OA    1  /*Offer accepted */
6.  #define OR    0  /*Offer rejected*/
7.
8.  mtype = {Offer, Response}
9.
10. chan S2P = [1] of {mtype, int};
11. chan P2S = [1] of {mtype, byte};
12.
13. /***Suppliers FSM***/
14. proctype Supplier()
15. {
16.
17.  int offerValue;
18.  byte responseValue; /*OA or OR*/
19.
20.  SupEContractSigned:
21.
22.  EditingOffer: /*under this label/state, we must take into account*/
          /*All possible actions that the supplier could take*/
```

```
23. if
24. :: offerValue = 30; /* An offer that is too high > MA*/
25. :: offerValue = 20; /* < MA */
26. :: offerValue = 10; /* < MA */
27. fi;
28.
29. if
30. :: S2P!Offer(offerValue) -> goto WaitingForResults;
31. :: skip; /*Taking into account the possiblity that*/
32. fi;       /*the supplier might not send anything */
33.
34. WaitingForResults:
35
36. if
37. :: P2S ? Response(responseValue);
38. :: timeout -> goto Dispute; /*if no response is received after 5 days*/
39. fi;
40.
41. if
42. :: (responseValue == OR) -> goto EditingOffer;
43. :: (responseValue == OA) -> goto Deal;
44. /*:: else -> printf("\n\n Error\n\n");*/
45. fi;
46.
47. Deal:
48. printf("\n\n Supplier: Deal \n\n");
59. goto end;
60.
61. Dispute:
62. printf("\n\n Dispute!!\n\n");
63. goto end;
64.
65. end:
66. printf("\n\n Supplier: End \n\n");
67
68. }
69.
70. /***Purchasers FSM***/
71. proctype Purchaser()
72. {
73. int  offerValue;
74. PurEContractSigned:
75.
76. WaitingForOffer:
77.
78. if
79. ::S2P ? Offer(offerValue)
80. ::timeout -> goto end
81. fi;
82.
83. DecidingToBuy:
84.
85. if
86. ::(offerValue>MA)-> P2S!Response(OR); goto WaitingForOffer;
87. :: else -> P2S ! Response (OA); goto Deal;
```

```
88.  fi;
89.
90.  Deal:
91.  printf("\n\n Purchaser: Deal\n\n");
92.
93.  end:
94.  printf("\n\n Purchaser: End\n\n");
95. }
96.
97.  init
98.  {
99.   run Supplier();
100.  run Purchaser();
101. }
```

The *Promela* code presented here is essentially the same code as in the previous section, but with inserted timeout constraints. This should correct the ambiguities that we detected in Section 4.7.1, and indeed this is verified after we run the Spin verifier once more, see figure 4.12.

Figure 4.12 shows that *Spin* did not detect any inaccuracies with the verification model with respect to the correctness requirements chosen (Invalid Endstates, Unreachable Code). The model can be checked against many correctness requirements as the designer deems necessary. For example, the verification model can be modified to explicitly state that the price offered by the *Supplier* has not been accepted by the Purchaser if the price exceeds an agreed price *P*. We can then insert assertions (Section 4.6.1) in relevant sections of the verification model to ensure that the *Supplier* does not make an offer > *P* in the form; *assert(offerValue<=P)* .

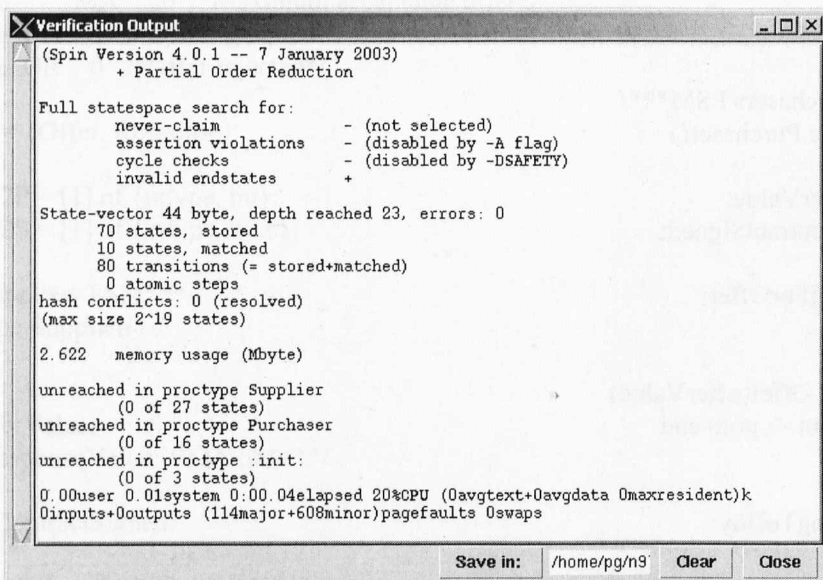


Fig.4.12 Verification output for the corrected verification model

4.8 Correctness requirements and Contracts, Discussion

In the previous section, we presented a contract, and illustrated how a verification tool can be utilized; to simulate possible interactions between the parties to the contract; and to verify the correctness of the FSM representation of the contract against some correctness requirements.

As we stated in Section 4.6 the choice of correctness requirements to verify any protocol is important. This applies also for the application of electronic contracts.

Consequently, what we wish to identify, is what correctness requirements may be required for the verification and the implementation of x-contracts. This will be investigated in depth in this section.

We can identify a number of correctness requirements that an x-contract must adhere to in order for it to be free of ambiguities, and therefore implementable, we can summarise these as follows:

(A) An x-contract must have clear end states. The person/s responsible eventually for implementing an x-contract must have a contract that explicitly defines what the valid end states are. That is what are the acceptable situations (triggered perhaps by Rights and/or Obligations) under which an x-contract may be terminated.
Definition of proper end states for x-contracts is necessary. This is to prevent deadlock, a situation where an execution sequence terminates at an unexpected "improper" end state. Under the *Promela* modelling language, the definition of a proper end state is as follows: Every process that was instantiated has either terminated or has reached a state marked as a proper end-state (See Section 4.6.2).

Notice that the *proctypes* for the *Purchaser* and the *Supplier*, in the validation models in Section 4.7.2 finish with "end" states. The Spin Validator implicitly detects any improper end-states.

Task/Requirement Summary: Identify all end states that are valid states for termination of the x-contract

(B) The x-contract may need to specify some essential "progress" states that the parties must go through during infinitely cycling transactions. I.e. the verification model of the x-contract, cannot infinitely cycle through states that are not labelled "progress-states". For example for the contract clause: Before the Supplier despatches the goods, he/she *must* receive the payment. This makes the "payment" state essential for the progress of the x-contract, and the verification model of the x-contract must not infinitely cycle through other states without going through the "payment" state.

We can express this in *Promela* by preceding the "progress" state with the label *progress*. In *Promela*, the execution sequences that violate this correctness claim are called *non-progress* cycles. The *Spin* verifier will when requested detect *non-progress* cycles.

Task/Requirement Summary: Identify all states within infinite cycles that are essential for the progress of the x-contract.

Example: In the x-contract of Section 4.7.2 we want to ensure that the *Purchaser* does not accept or reject an offer before an offer is actually made by the *Supplier*. The "receipt of an offer" qualifies as a progress state because (a) The *Purchaser* must rely on its occurrence to be able to make a decision, and (b) because this state can and should occur, infinitely often, if the *Supplier* continues to make offers (The repeated occurrence of this as we know means that the *Supplier* is repeatedly making unacceptable offers).

Verification:

We can modify the *Promela* code of Section 4.7.2 as follows:

```
WaitingForOffer:
if
::S2P ? Offer(offerValue);
  progressOnOffer: skip
::timeout -> goto end
fi;
```

We have inserted the progress label *progressOnOffer* just after waiting for the receipt of the offer state *S2P ? Offer(offerValue)* . Now after running the Syntax check, we can set the verification parameters to detect non progress cycles, see figure 4.3, and then click on the *Run* button. The results are presented in figure 4.13.

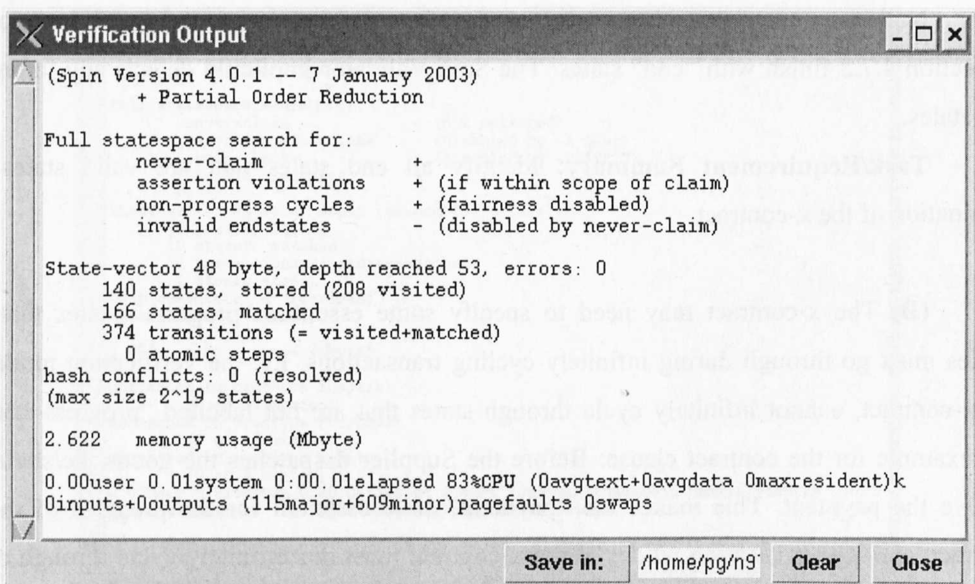


Fig.4.13 Verification Output for detection of non-progress cycles

If we compare figure 4.13 with figure 4.12, we notice under the "Full statespace search for" section that the safety requirement "invalid endstates" has been switched of for the purpose of this verification run, and we are checking for the "non-progress cycles" requirement.

The results show that there are "0" errors, so we can be sure that the "receipt of an offer" state is indeed a progress state as we require.

(C) The x-contract may need to specify situations or actions by the signatories of the contract that may not be executed infinitely often.

In the example presented in Section 4.7.2 a situation may occur where the Supplier can infinitely make the same unacceptable offer of Price>20. This reoccurring situation in *Promela* terms is what is known as *Livelock*, see Section 4.6.3, and this property, can be expressed using acceptance-state labels. An acceptance state label is any label starting with the character sequence "accept".

Task/Requirement summary: Identify states that must not be repeated infinitely often.

Example: let us take the situation we just presented: In our example of Section 4.7.2, we would not desire a situation where the supplier infinitely often makes undesirable offers. Therefore the state at which the Supplier is making unacceptable offers qualifies as an acceptance state.

Verification: We can modify the *Promela* code of Section 4.7.2 as follows:

```
EditingOffer: /*under this label/state, we must take into account*/
            /*All possible actions that the supplier could take*/
if
:: offerValue = 30;
  acceptOfferTooHigh: skip  /* An offer that is too high > MA*/
:: offerValue = 20; /* < MA */
:: offerValue = 10; /* < MA */
fi;
```

We have inserted the accept label *acceptOfferTooHigh* just after the *Supplier* edits an offer *offerValue = 30*. By inserting this label, we are telling the *Spin* verifier that we would not like the possibility to arise where the supplier may infinitely make an offer that is too high. After running the Syntax checker, we can set the verification parameters to detect non acceptance cycles, see figure 4.3, and then click on the *Run* button. The results are presented in figure 4.14.

```
Verification Output
pan: acceptance cycle (at depth 3)
pan: wrote pan_in.trail
(Spin Version 4.0.1 -- 7 January 2003)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (not selected)
        assertion violations    +
        acceptance   cycles     + (fairness disabled)
        invalid endstates       +

State-vector 44 byte, depth reached 26, errors: 1
      72 states, stored (81 visited)
       8 states, matched
      89 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

2.622   memory usage (Mbyte)

0.00user 0.02system 0:00.01elapsed 166%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (117major+608minor)pagefaults 0swaps
Save in: /home/pg/n9   Clear   Close
```

Fig.4.14 Verification output for detection of livelock (non-accept cycles)

The output results tell us that the process was not completed, and that the search was stopped at an error. As in Section 4.7.2, we will use *Spin* to trace the source of the error. Whenever Spin detects a correctness requirement violation, the window in figure 4.9 will always appear, and we can run a guided simulation through the route in which *Spin* detected the error. We know in this case where the error is, because we only have one accept label. Using the simulator is useful in complex x-contracts where we want to detect the source of an error between many accept labels. Going back to our example, after running the guided simulation, we can see the simulation output results in figure 4.15.

```
Simulation Output
preparing trail, please wait...done
  1:    proc  0 (:init:) line  92 "pan_in" (state 1)   [(run Supplier())]
  2:    proc  0 (:init:) line  93 "pan_in" (state 2)   [(run Purchaser())]
  3:    proc  1 (Supplier) line  25 "pan_in" (state 1)   [offerValue = 30]
<<<<<START OF CYCLE>>>>>
  4:    proc  1 (Supplier) line  26 "pan_in" (state 2)   [(1)]
  5:    proc  1 (Supplier) line  32 "pan_in" (state -)   [values: 1!Offer,30]
  5:    proc  1 (Supplier) line  32 "pan_in" (state 7)   [S2P!Offer,offerValue]
  6:    proc  2 (Purchaser) line  71 "pan_in" (state -)  [values: 1?Offer,30]
  6:    proc  2 (Purchaser) line  71 "pan_in" (state 1)  [S2P?Offer,offerValue]
  7:    proc  2 (Purchaser) line  72 "pan_in" (state 2)  [(1)]
  8:    proc  2 (Purchaser) line  79 "pan_in" (state 7)  [((offerValue>20))]
  9:    proc  2 (Purchaser) line  79 "pan_in" (state -)  [values: 2!Response,0]
  9:    proc  2 (Purchaser) line  79 "pan_in" (state 8)  [P2S!Response,0]
 10:    proc  1 (Supplier) line  39 "pan_in" (state -)   [values: 2?Response,0]
 10:    proc  1 (Supplier) line  39 "pan_in" (state 12)  [P2S?Response,responseValue]
 11:    proc  1 (Supplier) line  44 "pan_in" (state 17)  [((responseValue==0))]
 12:    proc  1 (Supplier) line  25 "pan_in" (state 1)   [offerValue = 30]
spin: trail ends after 13 steps
#processes: 3
 13:    proc  2 (Purchaser) line  70 "pan_in" (state 5)
 13:    proc  1 (Supplier) line  26 "pan_in" (state 2)
 13:    proc  0 (:init:) line  94 "pan_in" (state 3)
3 processes created
Exit-Status 0
Single Step   Suspend   Save in: sim.out   Clear   Cancel
```

Fig.4.15 Simulation output for path with livelock

As we can see, the trail ends after the Supplier makes an offer = 30, where we placed the *acceptOfferTooHigh* label in the *Promela* verification model.

We now know that we have an undesirable situation where the *Supplier* can make unacceptable offers infinitely. The contract does not deal with this situation either because: (1) for some reason the signatories have agreed to allow this. (2) They have simply omitted to explicitly insert detail that they may for a conventional contract (non x-contract) have taken for granted.

As the second possibility is the most probable, the following clause will have to be modified:

If no offer is sent within seven days after the signature of the x-contract, or after the latest rejected offer, the x-contract shall be terminated.

There are many solutions; one would be to limit the Supplier to $N<=10$ offers (for example). And the verification model will be modified accordingly.

(D) Transactions between parties to a contract may need to run in a certain sequence, and/or under certain conditions, depending on the specific requirements of the signatories.

An x-contract must define when each of the Rights and Obligations can be performed. Examples; an x-contract must define if payment for the goods is to be made before or after delivery of the goods; the FSM of an x-contract cannot reach a state where a complaint about the quality of the goods is sent before reaching the state where the goods are delivered. Some situations such as the later may seem obvious and may not be scrutinised in conventional contracts, but they must be clearly stated if a contract is to be enforced electronically.

Validation of this correctness requirement using *Promela* can be achieved through the use of "temporal claims" (Section 4.6.4). Temporal requirements can be very complex. They can even expand to include non-progress, and non-acceptance correctness requirements.

Task/Requirement summary: Identify the required sequencing of events/states for the correct performance of the contract, and identify the conditions (if any) under which this sequencing must occur.

Example: Let us take the correctness requirement that a complaint about the quality of the goods must not be sent by the *Purchaser* before the goods are received from the *Supplier*. We can present a verification model to express possible scenarios:

```
/* Goods complaint example
 *
 */

bool placeOrder = false ;
bool complaintRecd = false ;
bool orderRecd = false ;

active proctype OrderPlacement()
{
 // place an order
 placeOrder = true ;

 // Premature complaint randomly sent or not
 // sent to the Supplier

 if
 :: complaintRecd = true ; //complaint received by Supplier
 :: skip ;
 fi;

 // receive order
 orderRecd = true ;

 // a complaint was made
 if
 :: complaintRecd = true ;
 :: skip ;
 fi
}
```

Using *XSpin*, verification of temporal claims is done using the Linear Temporal Logic (LTL) Manager. See figure 4.16. We shall use the LTL Manager to check if sending a complaint before receiving the goods is a possible scenario of the above verification model. So looking at our verification model, we are claiming the following:

[] (placeOrder -> !complaintRecd U orderRecd) meaning:

It is invariantly true that following the placement of an order a complaint should NOT be received before the order is received. After this formula is entered into the LTL Manager we first generate the "Never Claim" by clicking the "Generate" button. After this, we can Run the verification, See figure 4.16.

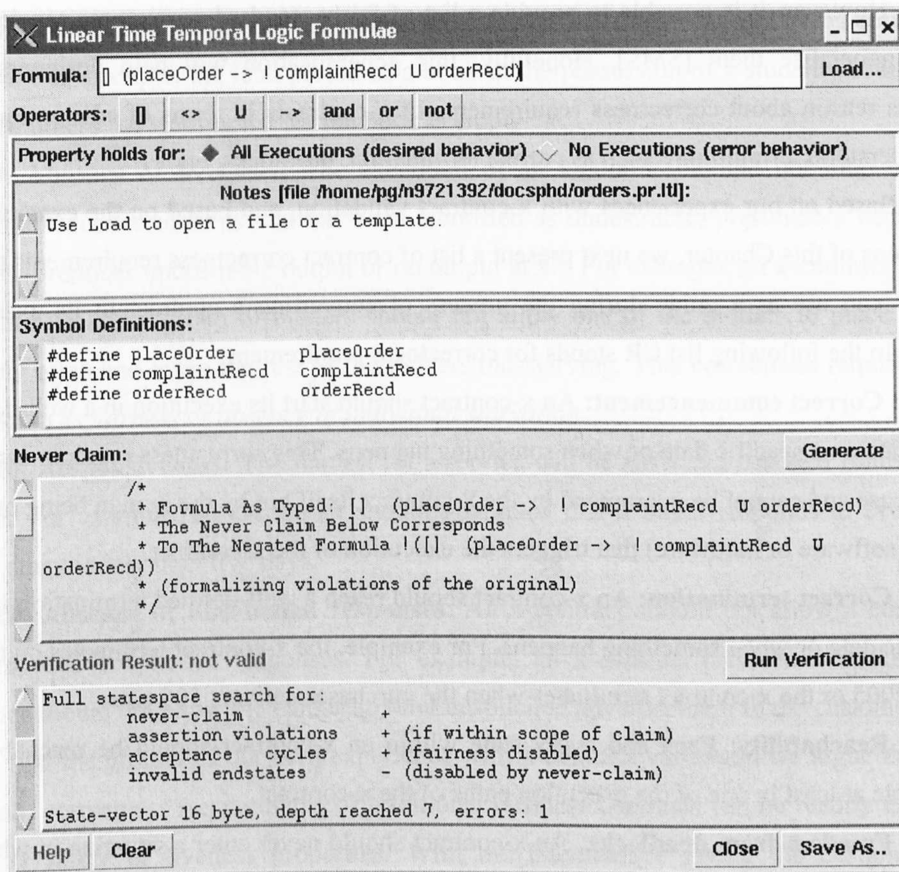


Fig.4.16 The LTL Manager

As we can see, and as expected, the verifier detects that our claim with respect to the above verification model is false. Again and as with previous examples, we get the window in figure 4.9 giving us the choice of a simulation through the erroneous path.

If we comment out/remove the section that gives the Purchaser the option to complain before receiving the order, the above Verification result becomes "Valid".

4.9 Common correctness requirements

Knowing the correctness requirements of an x-contract at design time is crucial as an x-contract can be proven correct only with respect to a specific list of correctness requirements. The parts of a contract that more likely contain logical inconsistencies vary from contract to contract. On the other hand, it is sensible to think, that different contract users would be interested in being assured of the correctness of different parts of a given contract. Because of this, it is too ambitious to intend to identify a complete list of correctness requirements for business contracts.

However, it is possible to provide a list of fairly standard correctness requirements and to generalise them [SMS]. Hopefully, this generalisation will help designers of x-contracts reason about correctness requirements of x-contracts in terms of conventional and well understood terminology such as correct termination, deadlocks, etc.

Based on our experiences with x-contract validation, and based on the examples and discussions of this Chapter, we next present a list of contract correctness requirements that the designer can use as a guide during the process of converting a contract to its executable version. In the following list CR stands for correctness requirement:

CR1: Correct commencement: An x-contract should start its execution in a well-defined initial state on a specific date or when something happens. This correctness requirement is a special case and cannot be guaranteed by the x-contract itself but by the human being or system (software or hardware) that triggers the execution of the x-contract.

CR2: **Correct termination:** An x-contract should reach a well-defined termination state on a specific date or when something happens. For example, the x-contract terminates on the 31st of Dec 2005 or the x-contract terminates when the purchaser delivers 500 cars.

CR3: **Reachability**: Each and every state within an x-contract should be reachable, i.e. executable at least in one of the execution paths of the x-contract.

CR4: **Freedom from deadlocks:** An x-contract should never enter a situation in which no further progress is possible. For example, an x-contract should not make a supplier wait for a payment before sending an item to the purchaser while the purchaser is waiting for the item before sending the payment to the supplier.

CR5: **Partial correctness:** If an x-contract begins its execution with a precondition true then, the x-contract will never terminate with the precondition false, regardless of the path followed by the x-contract from the initial to its final state. For example, if the amount of money borrowed by a customer from a bank is *Debt= 0* at the beginning of the x-contract, the x-contract cannot be closed unless *Debt=0.*

CR6: **Invariant:** If an x-contract begins its execution with a precondition true then, the precondition should remain true for the whole duration of the contract. A slight variation of this correctness requirement would be a requirement that the precondition remains true only or at least during certain parts of the execution of the x-contract. To mention an example we can think that an x-contract between a banker and a customer stipulates that the amount of money borrowed by the customer should never exceed the customer's credit limit.

CR7: **Occurrence or accessibility:** A given activity should be performed by an x-contract at least once no matter what execution path the x-contract performs. A slight variation of this requirement is one that demands that a certain activity should be performed infinitely often. For example, an x-contract between a bank and a customer should guarantee that the customer will receive bank statements at least once a month.

CR8: **Precedence:** An x-contract can perform a certain activity only if a given condition is satisfied. For example, the lend period of a book in the possession of a student should not be extended unless the waiting list for the book is empty.

CR9: **Absence of livelocks:** The execution of an x-contract should not loop infinitely through a sequence of steps that has been identified as undesirable, presumably because the sequence produces undesirable output or no output at all. For example, an x-contract between an auctioneer and a group of bidders should not allow one of the bidders to place his bids infinitely often and leave the rest of the bidders bid-starving. This correctness requirement is also known as *fairness* or *absence of individual starvation.*

CR10: **Responsiveness:** The request for a service will be answered before a finite amount of time. For example, an x-contract should guarantee that a buyer responds to every offer from a client in less than five days.

CR11: **Absence of unsolicited responses**: An x-contract should not allow a contractual party to send unsolicited responses. For example, an x-contract between a banker and a customer should not allow the banker to send unsolicited advertisement to the customer.

On the ground of our own experience with x-contract validation we argue that most, if not all, correctness requirements of traditional business contracts can be readily expressed either as safety or liveness properties. With the intention of giving the designer of an electronic contract some guidance about the kind of correctness requirement he/she is faced with, we will classify into safety and liveness properties the list of typical correctness requirements of electronic business contracts provided above:

- Safety properties: reachability, partial correctness, invariant, deadlocks, precedence, absence of unsolicited responses.
- Liveness properties: correct termination, occurrence, livelocks, responsiveness.

We further categorize Safety properties into, *general* safety properties that must hold true for any x-contract (CR3: Reachability, CR4: Freedom from deadlocks, CR11: Absence of unsolicited responses), and *specific* safety properties that must hold true only if so required by the contracting parties for the specific requirements of a certain x-contract (CR5: Partial correctness, CR6: Invariant, and CR8: Precedence).

Running the Spin validator under its default settings will check for *general* safety properties. Validation of the remaining *specific* safety properties can be done by inserting "Assertions" within the Promela code.

We are aware that it has been shown that not all correctness requirements can be readily classified as either safety or liveness property [NC00]. Contracting parties may desire complex correctness requirements that are a combination of a number of the above requirements. Fortunately, it has been formally proven that any correctness property can be

represented as the intersection of a safety property and a liveness property [AS85]. The idea behind our approach is that a complex correctness requirement demanded by a signing party can always be expressed as a combination of a number of the basic correctness requirements listed in Section 4.9.

4.10 Summary

It is crucial that we identify and eliminate the ambiguities that exist within the clauses of a text contract before it can be implemented electronically.

In this chapter, we have introduced the protocol modelling language Promela, and the protocol verification tool Spin. We have analysed with the aid of simple examples the correctness properties that must be satisfied for a contract to be correct. Based on our analysis we have developed a list of correctness requirements that we suggest that x-contract designers use during the contract validation process.

In the next chapter, we will test a number of example x-contracts for correctness claims that cover the above correctness requirements in more depth.

Chapter Five

Validation of Electronic Contracts: Examples

We present three different examples of text based documents (contracts) containing rules that govern the interaction between different parties. Through these examples, we demonstrate ideas developed in Chapter 3, and Chapter 4.

Our aim is to convert the text based contracts into executable contract models through a process that removes the ambiguities that may exist in the original text contracts. This is to facilitate the correct implementation of the x-contracts at run time.

There are many examples, where the interaction between two or more parties, over a network, calls for a set of rules that can be implemented to police this interaction.

In cases where the rules of interaction need to be negotiated and agreed upon by the interacting parties, the rules constitute "contract clauses", which will combine to form a contract that the parties must sign. This case will be the bases for our first and second examples.

There are cases however where the interaction between the parties is governed by rules that are already in place. The parties need only to understand them and agree upon them before the interaction can begin. Our third example reflects this case. We present the scenario where two or more parties are involved in a game that is played over the Internet. We use our third example to demonstrate interactions that involve more than two parties.

5.1 Contract for the supply of electronic goods

Our first example is a contract for the purchase and supply of e-goods. The contract we present is inspired from a contract in a paper written by GoodChild et al [GM00].

Our goal is to convert the contract into an x-contract model, and detect and eliminate ambiguities that may hinder the correct electronic implementation of the x-contract.

We will first present the original text contract in full. After that we will extract the sets of rights and obligations for each of the signatories to the contract, and map them into finite state machines for the *Purchaser* and for the *Supplier*. The FSM's are used to model the contract. This model is then converted into the *Promela* verification language. Finally we will use the *Spin* verifier to verify the correctness of the x-contract against the correctness requirements discussed in Chapter 4.

5.1.1 The Contract

This Deed of Agreement is entered into as of the Effective Date identified below.

BETWEEN

[Name] of [Address] (To be known as the (Supplier))	AND:	[Name] of [Address] (To be known as the (Purchaser))

WHEREAS (Supplier) desires to enter into an agreement to supply (Purchaser) with [Item] (To be known as (e-goods) in this Agreement). **NOW IT IS HEREBY AGREED** that (Supplier) and (Purchaser) shall enter into an agreement subject to the following terms and conditions:

1. Definitions and Interpretations
1.1 Price, Dollars or $ is a reference to the currency of the [Country].
1.2 All information (purchase order, payment, notifications, etc.), is to be sent electronically.
1.3 This agreement is governed by [Country] law and the parties hereby agree to submit to the jurisdiction of the Courts of the [Country] with respect to this agreement.

2. Commencement and Completion
2.1 The commencement date is scheduled as [date].
2.2 The completion date is scheduled as [date].
2.3 The schedule may be modified by agreement as defined in Section 9.

3. Purchase Orders
3.1 The (Purchaser) **shall** follow the (Supplier) price lists.
3.2 The (Purchaser) **shall** present (Supplier) with a purchase order for the provision of (E-goods) within 7 days of the commencement date.
3.3 The (Supplier) **shall** notify the (Purchaser) of acceptance or rejection of the purchase order within 7 days after the receipt of the purchase order.
3.4 If the purchase order is rejected, the (Purchaser) **shall** correct the purchase order within 14 days after the receipt of the notification.

4. Delivery
4.1 The delivery of the (e-goods) is the responsibility of the (Purchaser). The (Supplier) **shall** keep the E-good available for downloading at the specified e-address for at least 14 days after sending notification of acceptance of payment. The (Purchaser) **shall** download the (e-goods) within this period of time.

5. Payment
5.1 The payment **shall** be sent in full to the (Supplier) within 7 days after receiving a notification of acceptance of the purchase order.
5.2 The (Supplier) **shall** notify the (Purchaser) of acceptance or rejection of the payment within 7 days after the receipt of the payment.

6. E-goods rejection

6.1 If the (e-goods) do not comply with the order or the (Supplier) does not comply with any of the conditions, then the (Purchaser) is, at his/her sole discretion, **entitled to** reject the (e-goods).
6.2 The (Purchaser) **shall** either (a) notify the (Supplier), of acceptance of the (e-goods), within 7 days after receiving them, or (b) return the (e-goods) to the (Supplier), within 7 days after receiving them.

7. Replacement and refund
7.1 The (Supplier) **may** use his/her discretion to replace the (e-goods) according to the invoice or refund any monies paid.
7.2 The (Supplier) **shall** either (a) notify the (Purchaser) of refusal to replace or refund, within 14 days after the receipt of the rejected (E-goods), or (b) replace or refund any monies paid, within 14 days after the receipt of the rejected (e-goods).
7.3 In the case of a dispute in which the (Supplier) refuses to provide a requested replacement or refund by the (Purchaser) within 14 days of the (Purchaser) returning rejected (e-goods), then the Purchaser **shall** terminate the contract.

8. Termination
8.1 If (Purchaser) or (Supplier), fail to carry out any of their obligations and duties under this agreement, the offended party **shall** terminate the contract.

9. Disputes
9.1 (Supplier) and (Purchaser) **shall** attempt to settle all disputes, claims or controversies arising under or in connection with the agreement through consultation and negotiations in good faith and a spirit of mutual cooperation.
9.2(Supplier) and (Purchaser) **shall** provide electronic evidences about breaches of the x-contract.
9.3 This method of determination of any dispute is without prejudice to the right of any party to have the matter judicially determined by a [Country] Court of competent jurisdiction.

10. Amendment
10.1 This agreement **may** only be amended in writing signed by or on behalf of both parties.

E-SIGNATURES
In witness whereof (Supplier) and (Purchaser) have caused this agreement to be entered into by their duly authorized representatives as of the effective date written below.

Effective date of this agreement: [day] of [month] [year]

[E-signature] [Person] [Role]	[E-signature] [Person] [Role]

E-address for Notices:

[E-address]	[E-address]

5.1.2 Split of rights and obligations

Before describing the x-contract in FSM notation, it is advisable to extract, from the English text, the purchaser's and supplier's Rights (R) and Obligations (O).

Supplier's obligations

O01: Notify Purchaser of acceptance or rejection of Purchase order within 7 days after receipt of purchase order.

O02: Notify Purchaser of acceptance or rejection of the payment within 7 days of the Supplier receiving it.

O03: Place e-goods at e-address for 14 days after sending a notification of acceptance of payment.

O04: Either (a) Notify the Purchaser of rejection of a remedy request within 14 days after receipt of the rejected e-goods, or (b) provide a remedy within 14 days after receipt the rejected e-goods.

O05: Provide electronic evidences of breach of the x-contract.

O06: Terminate the x-contract if the Purchaser is in breach of contract.

Supplier's rights

R01: Reject or accept a purchase order.

R02: Reject or accept a payment.

R03: Either (a) Refuse to remedy within 14 days after receipt of the rejected e-goods, or (b) Accept to remedy within 14 days after receipt the rejected e-goods.

R04: Amend contract but only in agreement with the Purchaser.

Purchaser's obligations

O01: Follow the supplier's price lists.

O02: Present a purchase order within 7 days of the commencement date.

O03: Correct a Purchase order within 14 days after receipt of a notification of the first rejection of the Purchase order.

O04: Send full payment within 7 days after receiving notification of acceptance of purchase order.

O05: Download the e-good/s within 14 days after the receipt of the acceptance of the payment.

O06: Send acceptance or rejection of e-goods within 7 days of receiving them.

O07: Provide electronic evidences of breach of x-contract.

O08: Terminate the x-contract if either (a) The Supplier is in breach of contract, or (b) In the case of a dispute where the Supplier does not provide replacement or remedy for rejected e-goods.

Purchaser's rights

R01: Reject e-goods that fail to match the description/requirement.

R02: Amend contract but only in agreement with the supplier.

Next we map the Rights and Obligation above into 2 finite state machines, one for the Purchaser, and one for the Supplier.

5.1.3 The finite state machines (The x-contract model)

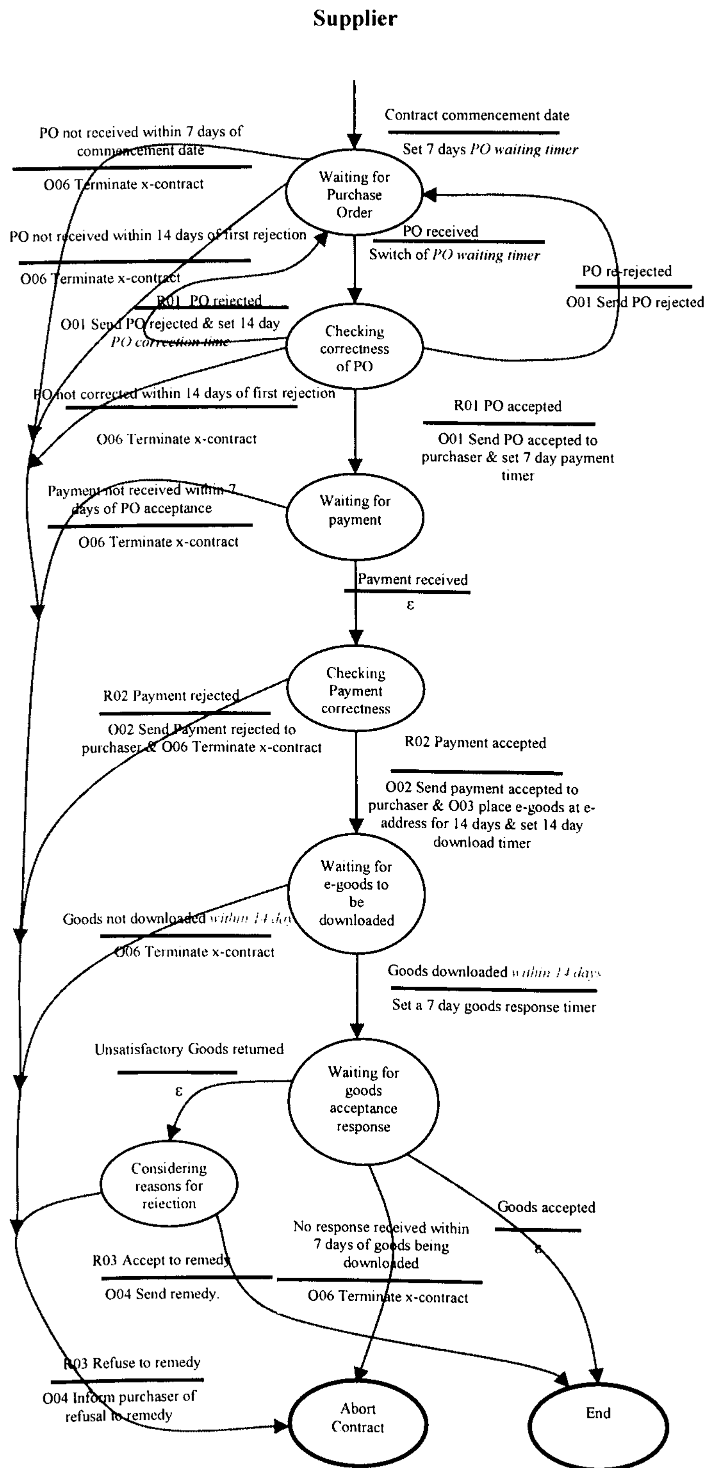


Fig. 5.1. Supplier's FSM.

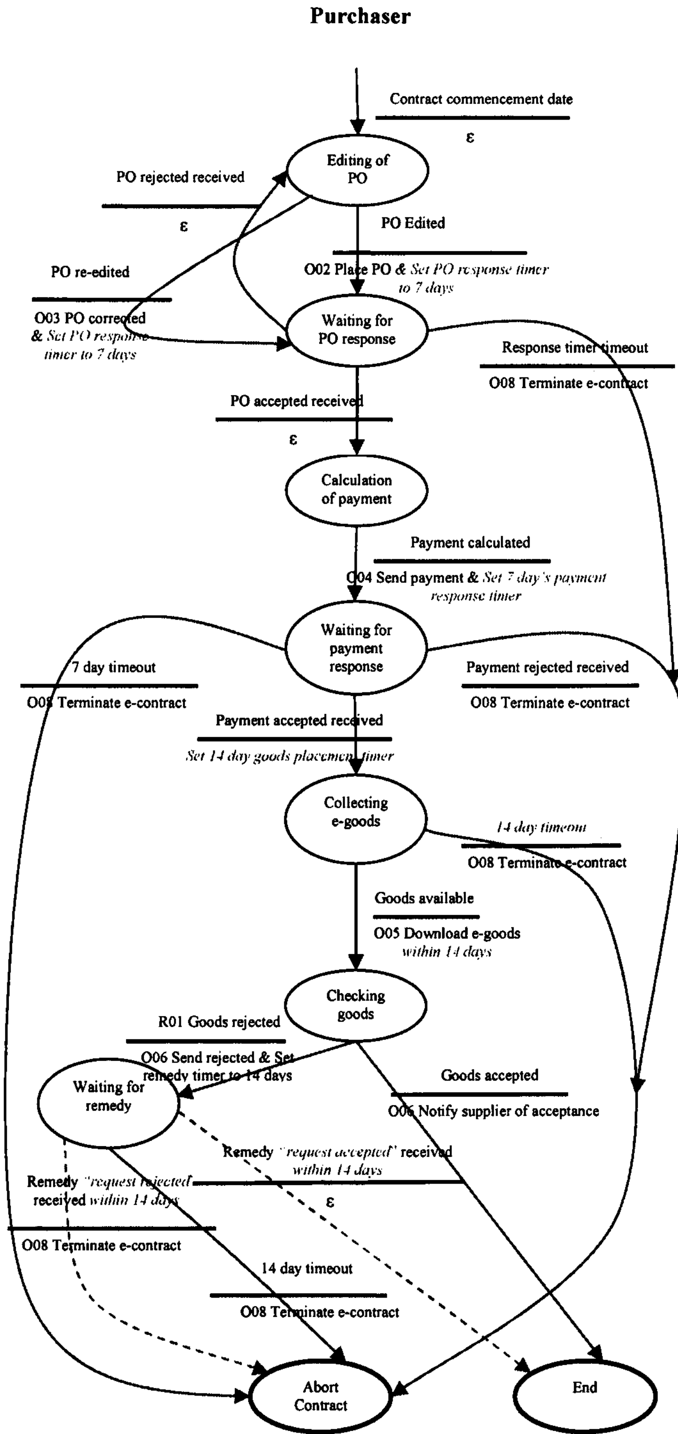


Fig. 5.2. Purchaser's FSM.

In figure.5.1 and figure 5.2, PO stands for Purchase Order.

5.1.4 The Verification model

Our next task is to represent the FSMs in figures 5.1 and 5.2, in the modeling language – *Promela-* that is the input language of the verifier *Spin*. The complete verification model is presented next:

```
1   /*  X-contract model for the purchase and supply of e-goods.
2    *
3    *
4    *   Programme created using XSPIN for simulation and
5    *   verification of FSM correctness.
6    *
7    *   AUTHOR: Ellis Solaiman
8    *   University Of Newcastle  Upon Tyne
9    *   Date of Creation     12 December 2002
10   *   Last Update          18 July     2003
11   *
12   */
13
14  //Definition of constants
15  #define Terminated 5
16
17  #define Yes 1
18  #define No  0
19
20  #define Good 1
21  #define Bad 0
22
23  #define Accept 1
24  #define Reject 0
25  #define SReject 2 /*this for a second reject*/
26
27  #define correct 1
28  #define incorrect 0
29
30  #define Available 1
31  #define NotAvailable 0
32
33  // Messages that will be passed between the purchaser and
supplier
34  // PO is the Purchase Order
35  mtype = {PO, Payment, Download, Response, GoodsPlaced,
Remedy,Goods,Check,Rrequest}
36
37  // Channels of communication
38  chan P2S=[1] of {mtype, byte}; // channel purchaser to supplier
39  chan S2P=[1] of {mtype, byte}; // channel supplier to purchaser
40
41  byte goodsPlaced = No; // Have the goods been placed for
download?
42  byte downloadInTime = No; // Have the goods been downloaded in
time?
43
44
45  /****** SUPPLIER FSM ******/
46  proctype Supplier()
47  {
```

```
48   int  poVal;  //Good PO or Bad PO
49   byte payVal = Bad;
50   byte remedyChoice;
51   byte goods;
52
53
54  /*** Waiting for Purchase Order ***/
55  WaitingforPO:
56
57  if //This if fi statement deals with both the 7 day and 14 day
timer*/
58  ::P2S ? PO(poVal)
59  ::timeout -> goto Terminate
60  fi;
61
62  /*** Checking correctness of PO ***/
63  CheckPO:
64
65  if
66  ::if
67    ::(poVal==Good)-> S2P ! Response(Accept); goto
WaitingforPayment
68    ::(poVal==Bad)-> S2P ! Response(Reject); goto WaitingforPO
69    fi;
70  :: skip;   //The Supplier fails/forgets to send a response in
time
71  :: goto Terminate;   //Supplier aborts x-contract
72  fi;
73
74  /*** Waiting for Payment ***/
75  WaitingforPayment:
76
77  if
78  ::P2S ? Payment(payVal)
79  ::timeout -> goto Terminate //Payment not received in time
80  fi;
81
82  /***Checking Payment Correctness***/
83  CheckingPayment:
84
85  if
86  ::if
87    ::(payVal == Bad)-> S2P ! Response(Reject); goto Terminate;
88    ::(payVal == Good)-> S2P ! Response(Accept); goto PlaceGoods;
89    fi;
90  :: goto Terminate; // Supplier aborts contract
91  :: skip; //Supplier forgets to respond
92  fi;
93
94  /*** Place goods to be downloaded ***/
95  PlaceGoods:
96  if                    //Safety claim 1: Payment must be correct
before delivery
97  :: goodsPlaced = Yes; assert (payVal == Good);
98  :: goodsPlaced = No;
99  fi;
100
101 /*** Waiting for e-goods to be downloaded ***/
102 WaitingforDownload:
103
104 if
```

```
105 ::(downloadInTime==Yes) //goods downloaded in time
106 ::(downloadInTime==No)-> goto Terminate;
107 fi;
108
109 /*** Purchasers Response to the Goods ***/
110 GoodsResponse:
111
112 //Safety claim 2: Asserting that the Supplier does
113 //not proceed if the goods are not downloaded in time
114 assert(downloadInTime == Yes);
115
116 if
117 ::P2S ? Goods(goods)  // Complaint made, and goods returned in
time
118 ::timeout ->goto end; // No complaint made within the time limit
119 fi;
120
121
122 if
123 :: (goods == Good)  -> goto end;
124 :: (goods == Bad) -> goto RemedyConsider;
125 fi;
126
127 /*** Considering reasons for rejection of goods ***/
128 RemedyConsider:
129
130 if
131 :: remedyChoice=Yes;
132 :: remedyChoice=No;
133 fi;
134
135 S2P!Remedy(remedyChoice);
136 goto end;
137
138 /*** Abort x-contract ***/
139 Terminate:
140
141 printf("\n\nUnsatisfactory Termination\n\n");
142 goto fin;
143
144 /*** Deal reached ***/
145 end:
146
147 //Safety claim 3: The following must hold true if a deal is to be
reached
148 assert(poVal == Good);
149 assert(payVal == Good);
150 assert(downloadInTime == Yes);
151
152 printf("\n\nSupplier in Deal state\n\n");
153
154
155 /*** End of x-contract ***/
156 fin:
157 printf("\n\nEnd\n\n");
158
159 }
160
161
162
163
```

```
164 /****** PURCHASER FSM ******/
165
166 proctype Purchaser()
167 {
168
169 int poVal;          // Purchase Order sent
170 byte responseVal; // Response received
171 int poResponse;
172 byte pVal;          // Payment sent correct or incorrect
173 byte goods;         // Acceptable or non-acceptable Goods
174 byte remedy;        // acceptable or non-acceptable remedy
175
176
177 /*** Placement of Purchase Order***/
178 PlacementOfPO:
179
180 if
181 :: poVal = Good; // Randomly choose between sending a good PO
182 :: poVal = Bad;  // and a Bad PO
183 fi;
184
185 /*** Send the PO ***/
186 if
187 ::P2S ! PO(poVal);
188 ::goto Abort; //modeling the possibility that Purchaser might not
send a PO
189 fi;
190
191 /*** Waiting for the Supplier's Response to PO ***/
192 WaitingResponse:
193
194 if
195 ::S2P ? Response(poResponse);
196 ::timeout -> goto Abort; // Response not received in time
197 fi;
198
199
200 if
201 ::(poResponse == Accept) -> goto  PlacePayment // PO accepted
202 ::(poResponse == Reject) -> goto PlacementOfPO // PO not accepted
203 ::(poResponse == SReject) -> goto Abort;       // Final reject
204 fi;
205
206
207 /*** Make payment ***/
208 PlacePayment:
209
210 if
211 :: pVal = correct;   // Randomly choose between sending a correct
payment
212 :: pVal = incorrect; // and an incorrect payment
213 fi;
214
215 if
216 :: P2S ! Payment(pVal) ->
217 :: skip;       // Purchaser fails/forgets to make payment
218 :: goto Abort; // Purchaser aborts x-contract
219 fi;
220
221
222 /*** Waiting for Payment response ***/
```

```
223 if
224 ::S2P ? Response(responseVal) ->
225   if
226         ::(responseVal == Accept) -> goto CheckGoodsAvailability;
227   ::(responseVal == Reject) -> goto Abort;
228         fi;
229 ::timeout -> goto Abort;
230 fi;
231
232
233 /*** Check that goods are available for download ***/
234 CheckGoodsAvailability:
235
236 if
237 ::(goodsPlaced == Yes)->
238 ::(goodsPlaced == No) -> goto Abort; //Goods not available for 14
days after making payment*/
239 fi;
240
241
242 if // Random if/fi statement
243 ::downloadInTime = Yes
244 ::downloadInTime = No
245 fi;
246
247
248 /****Check goods****/
249 CheckGoods: /*added*/
250
251 if // Goods accepted or not accepted
252 :: goods = Bad;
253 :: goods = Good;
254 fi;
255
256 if
257 :: (goods == Bad) -> P2S ! Goods(Reject);  goto WaitforRemedy
258 :: (goods == Good) -> P2S ! Goods(Accept); goto end
259 fi;
260
261
262 /*** Wait for remedy ***/
263 WaitforRemedy:
264
265 if
266 ::S2P ? Remedy(remedy) -> assert(remedy==Yes);goto end;// Safety
claim 4
267 ::timeout -> goto Abort; // Remedy not recieved in time
268 fi;
269
270
271 /*** Abort x-contract ***/
272 Abort:
273
274 printf("\n\nUnsatifactory Termination\n\n");
275 goto fin;
276
277 /*** Deal state ***/
278 end:
279 //Safety claim 4: The following must hold true if a deal is to be
reached
280 assert(goodsPlaced == Yes);
```

```
281
282 printf("\n\nPurchaser in Deal State\n\n");
283
284
285 /*******End x-contract**********/
286 fin:
287 printf("\n\nEnd\n\n");
288
289 }
290
291
292 /******Initiate purchasers and suppliers processes**********/
293 init
294 {
295 run Purchaser();
296 run Supplier();
297 }
```

5.1.5 X-contract verification

General Safety properties

CR3: Reachability, CR4: Freedom from deadlocks, and CR11: Absence of unsolicited responses are the *general* safety properties against which any contract must be validated. It is crucial that the general safety requirements are passed by the x-contract model. This proves that the FSMs will execute correctly, and that they are correct entities, in that both of the FSMs for each of the signatories, can deal with any interactions that are passed between them including non expected ones (CR11), can deal with possible situations where messages are not being passed where they should be (CR4), and that all states and code within the FSMs are reachable at least in one of the many possible paths through the FSMs. Other correctness requirements whether *specific safety* properties, or *liveness* properties, will be tested for in the following sections, to verify whether the FSMs perform operations desired by the signatories.

We will therefore begin by validating the model against the general correctness requirements.

Figure 5.3 shows the general verification options being selected. We can now run the verifier. Spin initially detected a number of Deadlock violations in which both of the FSMs in different scenarios were trapped in certain states. This is because the contract did not have time limits in which some messages had to be passed between the FSMs. The contract and the FSMs were modified (the modifications have been inserted the FSMs using italics for text and dashed lines for the FSM state transition arrows), and we run the Spin verifier again. The results are presented in figure 5.4. As can be seen, the verifier detects no errors, so the model is correct with respect to the general safety correctness requirements CR3, CR4, CR11. Therefore our model is implementable as an x-contract. However the signatories may wish to

test the model for specific safety requirements before being confident of the correctness of the x-contract.

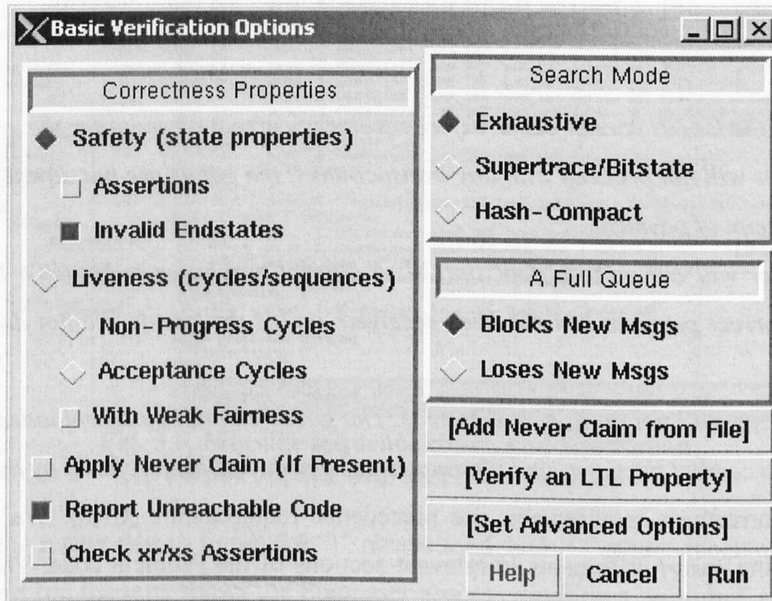


Fig.5.3. Selection of general safety requirements for verification

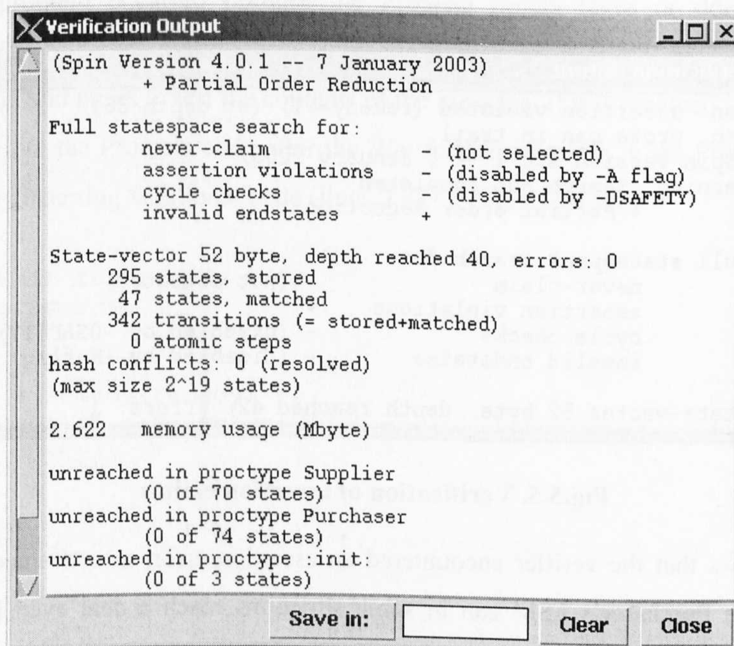


Fig.5.4. Verification output for general safety properties

Specific Safety properties

The specific safety requirements discussed in the previous chapter are: CR5: Partial correctness, CR6: Invariant, and CR8: Precedence. The signatories to the contract have requested to test the x-contract model against the following requirements:

1. Delivery of the Goods cannot occur before receipt of correct payment for the goods.

2. The Supplier will not proceed with any transactions if the goods are not downloaded within 14 days of receipt of payment.

3. The Supplier will not go to a deal state if: A correct purchaser order (PO) has not been received, a correct payment has not been received, and if the Goods are not downloaded in time.

4. The Purchaser will not go to a deal state if: The goods can not be downloaded for 14 days after making a correct payment, and if a requested remedy has been refused by the supplier.

All correctness requirements are precedence requirements (CR8). We will test for these by placing *assert* statements in relevant sections of the Promela code. The reader can see these in lines: 96, 114, 147, 266 and 280 of the Promela code listed in Section 5.1.4. Next we will initiate the verifier, and check the *assertions* option under *safety (state properties)*, see figure 5.3. Results from the verification can be seen in figure 5.5.

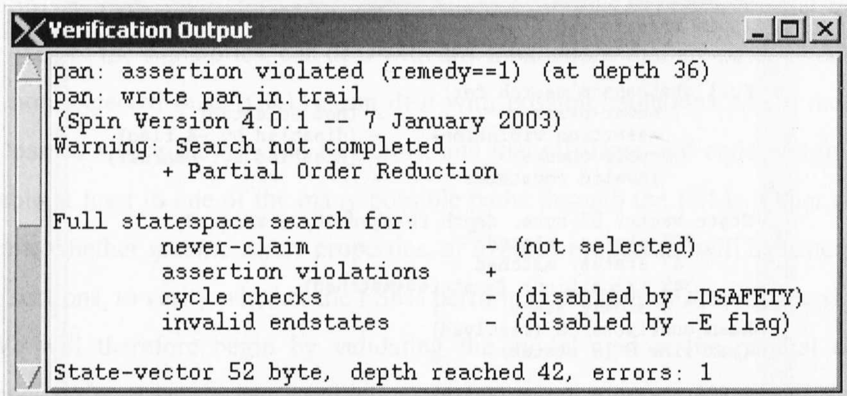


Fig.5.5. Verification of assertion claims

Figure 5.5 shows that the verifier encountered an assertion error, assert(remedy ==1). This tells us that the Purchaser's FSM can in some situations reach a deal even if a requested remedy was not granted by the Supplier, and therefore the verification model fails the above requirement 4. We can run the simulator through the erroneous path, the simulation output, and the message sequence chart are presented in figure 5.6.

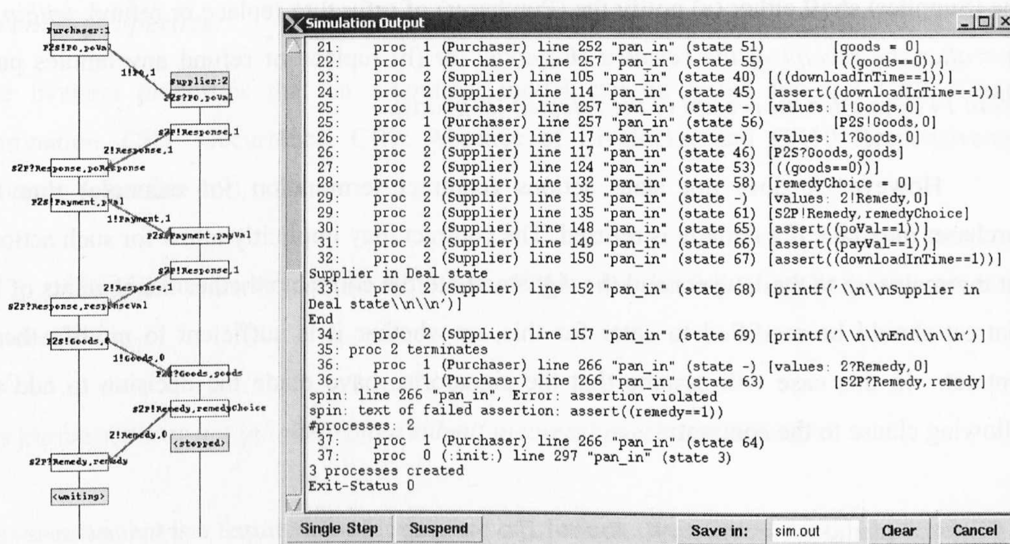


Fig. 5.6. Message sequence chart, and Simulation output of path with assertion violation

We remind the reader that in figure 5.6 "!" means send, and "?" means receive. The message sequence chart (on the left) shows the Purchaser process reaching a "waiting" state indicating a problem, and the simulation output clearly detects an assertion violation at step 37, indicating a problem in line 266 of the Promela code.

The problem occurred because the Promela model takes or does not take the Purchasers FSM to the deal state, based only on whether the Purchaser receives a remedy message or not, and omits to test the contents of the message, if it is received.

Therefore the Promela code after the WaitforRemedy state in the Promela code must be modified by inserting additional code (lines 270 – 273) as follows:

```
262  /*** Wait for remedy ***/
263  WaitforRemedy:
264
265  if
266  ::S2P ? Remedy(remedy) ->
267  ::timeout -> goto Abort; // Remedy not recieved in time
268  fi;
269
270  if
271  :: (remedy == Yes) -> assert(remedy==Yes); goto end ;//Safety
                                                              claim 4
272  :: (remedy == No) -> goto Abort
273  fi;
```

Meaning that if a remedy request has been accepted (remedy == Yes) then end the contract satisfactorily, and if a remedy request has been rejected, then abort the contract.

The relevant section (Section 7, Replacement and refund) of the original contract, states in section 7.2:

The (Supplier) **shall** either (a) notify the (Purchaser) of refusal to replace or refund, *within 14 days after the receipt of the rejected (E-goods),* or (b) replace or refund any monies paid, *within 14 days after the receipt of the rejected (E-goods).*

However it does not state actions (contract termination for example) that the Purchaser may take if a remedy is refused. The contract may implicitly allow for such actions, but it remains up to the lawyers and the signatories to the contract whether the contents of the contract should be modified to cater for this, or whether it is sufficient to modify the x-contract. In this case will assume that the signatories have made the decision to add the following clause to the contract:

7.3 In the case of a dispute in which the (Supplier) refuses to provide a requested replacement or refund by the (Purchaser) within 14 days of the (Purchaser) returning rejected (E-goods), then the Purchaser may terminate the contract.

The reason for this is that Section 8 of the contract allows for termination only in the case where the opposite party fails to perform a duty or an obligation, and does not allow for termination in the case of a dispute based on non obligatory actions. Therefore, the new clause 7.3 allows the Purchaser to terminate the contract in a special situation in which the parties disagree over replacement of rejected goods or refund.

After this modification, the validator detected a second assertion violation *"assert(payVal == 1);"* at line 97, caused by the *skip;* statement in line 91:

```
91 :: skip; //Supplier forgets to respond
```

This statement was inserted within the Supplier's FSM by the model designer, in order to test the ability of Purchaser's FSM to deal with a scenario that involves the Supplier forgetting to check the payment, and as a consequence forgetting to respond to the purchaser with acceptance or rejection of the payment. However, once this ability was established, the skip statement should have been removed, as it remains incorrect for the Supplier's FSM to allow him the possibility of not checking the correctness of the payment, and not responding to the Purchaser, even if the Purchaser's FSM can deal with this scenario. Therefore line 91 can simply be deleted. Also similar skip statements used for testing and forgotten throughout the Promela model must be removed.

After modifying the code, there are no safety violations.

Liveness properties:

The liveness properties that an x-contract model can be tested for are: CR2: Correct termination, CR7: Occurrence, CR9: Absence of Livelocks, and CR10 Responsiveness. (Chapter 4, Section 4.9).

The signatories would like to test the FSMs, for the following liveness correctness requirement:

The purchaser may not infinitely often submit incorrect purchaser orders.

This requirement is a requirement for absence of Livelock. We can test for this by inserting an *accept* label in the relevant section of the code as follows:

```
178 PlacementOfPO:
179
180 if
181 :: poVal = Good;
182 :: poVal = Bad; acceptBadOffer: skip
183 fi;
184
```

We will now run the validator after checking the "Acceptance cycles" option under "Liveness" in figure (BasicVerificationoptions).

```
Verification Output
pan: acceptance cycle (at depth 2)
pan: wrote pan_in.trail
(Spin Version 4.0.1 -- 7 January 2003)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim             (not selected)
        assertion violations    +
        acceptance   cycles     + (fairness disabled)
        invalid endstates       +

State-vector 52 byte, depth reached 42, errors: 1
```

Fig.5.7. Verifier detects Livelock

As we can see, figure 5.7, shows that there is indeed livelock. This tells us that it is possible the model will, through at least one sequence of execution, loop infinitely through states that allow the purchaser to submit incorrect purchase orders. We can run the simulator through the path in which the problem was detected (figure 5.8). From the simulator we realise that the problem is a programming error, caused by a presumption on the designers part that the "timeout" statement in the following code segment is sufficient to model the timeout

complexities required in the contract text. What is missing is a variable (*poTimer*) to model the 14 day time limit in which the purchaser must submit a correct purchase order:

```
55   /*** Waiting for Purchase Order ***/
56   WaitingforPO:
57
58   if //This if fi statement deals with both the 7 day and 14 day
         timer
59   ::P2S ? PO(poVal)
60   ::timeout -> goto Terminate
61   fi;
62
63   /*** Checking correctness of PO ***/
64   CheckPO:
65
66   if
67   :: if
68      ::(poVal==Good)-> S2P ! Response(Accept); goto
        WaitingforPayment
69      ::(poVal==Bad)->
70        if
71        :: (poTimer == 14) -> S2P ! Response(Reject); goto
             Terminate
72        :: (poTimer < 14 ) -> S2P ! Response(Reject); poTimer++;
             goto WaitingforPO
73        fi;
74      fi;
75   :: goto Terminate;     //Supplier aborts x-contract
76   fi;
```

```
Simulation Output
preparing trail, please wait...done
  1:     proc  0 (:init:) line 300 "pan_in" (state 1)     [(run Purchaser())]
  2:     proc  0 (:init:) line 301 "pan_in" (state 2)     [(run Supplier())]
<<<<<START OF CYCLE>>>>>
  3:     proc  1 (Purchaser) line 182 "pan_in" (state 2) [poVal = 0]
  4:     proc  1 (Purchaser) line 183 "pan_in" (state 3) [(1)]
  5:     proc  1 (Purchaser) line 188 "pan_in" (state -) [values: 1!PO,0]
  5:     proc  1 (Purchaser) line 188 "pan_in" (state 6) [P2S!PO,poVal]
  6:     proc  2 (Supplier) line  58 "pan_in" (state -)  [values: 1?PO,0]
  6:     proc  2 (Supplier) line  58 "pan_in" (state 1)  [P2S?PO,poVal]
  7:     proc  2 (Supplier) line  68 "pan_in" (state 9)  [((poVal==0))]
  8:     proc  2 (Supplier) line  68 "pan_in" (state -)  [values: 2!Response,0]
  8:     proc  2 (Supplier) line  68 "pan_in" (state 10) [S2P!Response,0]
  9:     proc  1 (Purchaser) line 196 "pan_in" (state -) [values: 2?Response,0]
  9:     proc  1 (Purchaser) line 196 "pan_in" (state 10)    [S2P?Response,poResponse]
 10:     proc  1 (Purchaser) line 203 "pan_in" (state 17)        [((poResponse==0))]
spin: trail ends after 11 steps
#processes: 3
 11:     proc  2 (Supplier) line  57 "pan_in" (state 4)
 11:     proc  1 (Purchaser) line 180 "pan_in" (state 4)
 11:     proc  0 (:init:) line 302 "pan_in" (state 3)
3 processes created
Exit-Status 0

Single Step   Suspend   Save in: sim.out   Clear   Cancel
```

Fig.5.8 Path through which Livelock was detected

After making the modifications to the code, the Spin validator detects no Livelock errors. This example shows how Spin can be useful in detecting programming errors, which if left unchecked could lead to problems at the time of implementation.

After testing the x-contract model against safety, and liveness correctness requirements, and correcting detected errors, the x-contract is ready for implementation.

5.2 Example of a contract for renting cars

In this Section, we present an example of a contract between a customer to be known as the *Renter*, and an international car rental company to be known as the *Owner*. The contract was chosen as an example of agreements that involve the provision of a service rather than the purchase and supply of goods. It is also an example that demonstrates extreme ambiguities and interpretation problems that may face a designer while constructing the x-contract model. As with the example in Section 5.1, this contract could be perceived to be of a B2B (Business to Business) or a B2C (Business to Customer) nature. The contact, which was taken from [LFK] is presented next.

5.2.1 Car Rental Contract

This Agreement is made and entered in this _(1)_ day of ________(2)________, 19_(3)_, between _________(4)_____________, of _________________(5)_____________________, hereinafter called "Owner", and __________(6)___________, of ______________(7)________________, hereinafter called "Renter".

Vehicle

The vehicle which the Owner hereby agrees to rent is:

_____(8)_______ _________(9)_________ _______(10)______

____(11)_______ ________(12)_________ _______(13)______

Mileage at beginning of rental period: ____(14)____

The Owner represents that to the best of his knowledge and belief that said vehicle is in sound and safe condition and free of any known faults or defects which would affect its safe operation under normal use.

Rental Period

The Owner agrees to rent the above-described vehicle to the Renter for a period of____(15)_____ beginning at _(16)_M. on ________(17)________ and ending at _(16)_M. on ________(18)________.

The Renter agrees (a) that the rented vehicle shall not be used to carry passengers or property for hire; (b) that the rented vehicle shall not be used to carry passengers other than in the interior or cab of the vehicle; (c) that the rented vehicle shall not be used to carry passengers in excess of the capacity thereof; (d) not to use the vehicle to push, propel or two another vehicle, trailer or any other thing without the written permission of the owner; (e) not to use the vehicle for any race or competition; (f) not to use the

vehicle for any illegal purpose; (g) not to operate the vehicle in a negligent manner; (h) not to permit the vehicle to be operated by any other person without the written permission of the owner; and (i) not to carry passengers, property or materials in excess of the rated weight carrying capacity of the vehicle.

Insurance

The Renter hereby agrees that he shall fully indemnify the Owner for any and all loss of or damage to the vehicle or equipment during the term of this Agreement whether caused by collision, fire, flood, vandalism, theft or any other cause, except that which shall be determined to be caused by a fault or defect of the vehicle or equipment.

Rental Rate

The Renter hereby agrees to pay the Owner at the rate of $___(19)___ per _____(20)_____ for the use of said vehicle. All fuel used shall be paid for by the Renter.

Deposit

The Renter further agrees to make a deposit of $____(21)____ with the Owner, said deposit to be used, in the event of loss of or damage to the vehicle or equipment during the term of this Agreement, to defray fully or partially the cost of necessary repairs or replacement. In the absence of damage or loss, said deposit shall be credited toward payment of the rental fee and any excess shall be returned to the Renter.

Return of Vehicle to Owner

The Renter hereby agrees to return said vehicle to the Owner at _________(22)__________ no later than ________(23)____________.

IN WITNESS WHEREOF, the parties hereto hereby execute this

Agreement on the date first above written.

_____________(24)______________

_____________(25)______________

Our aim is to eliminate any ambiguities that could exist is the contract so that it could be implemented electronically.

To achieve this, we will follow our familiarized sequence of steps. First we will extract the sets of rights and obligations from the contract, and then map them into finite state machines for the *Renter* and the *Owner*. After this we will code the finite state machines as a *Promela* verification model, and check it for ambiguities against the set of correctness requirements we conceived in Section 4.9.

One immediately noticeable ambiguity in the car rental contract is that if one of the parties fails to perform one or more obligations, the text of the contract does not specify what action the opposing party may take. This is an obvious ambiguity that was detected through manual inspection without requiring the Spin validator, and it can be fixed by adding additional clauses similar to the termination and dispute clauses in the contract in Section 5.1.1. We will presume that the signatories and their legal advisors have agreed to this, and we will proceed with adding the clauses:

Additional contract clauses:

```
Termination
If (Owner) or (Renter), fail to carry out any of their obligations
and duties under this agreement, the offended party may issue a
notice specifying the breach and terminate the contract.

Disputes
(Owner) and (Renter) shall attempt to settle all disputes, claims or
controversies arising under or in connection with the agreement
through consultation and negotiations in good faith and a spirit of
mutual cooperation.
(Owner) and (Renter) shall provide electronic evidences about
breaches of the contract.
This method of determination of any dispute is without prejudice to
the right of any party to have the matter judicially determined by
a [Country] Court of competent jurisdiction.
```

We will next extract the rights and Obligations from the text of the contract as well as the text of the additional clauses.

5.2.2 Parties' rights and obligations

Owners Obligations

OO1: Provide vehicle for period 'p'.

OO2: Provide vehicle in acceptable condition.

OO3: Return the deposit if vehicle is returned in acceptable condition.

OO4: Settle all disputes.

OO5: Provide electronic evidences about breaches of the contract.

Owners Rights

OR1: Terminate x-contract, if the Renter is in breach of the contract.

Renter's Obligations

RO1: Make deposit.

RO2: Make payments on time.

RO3: Follow vehicle use agreements ('a' to 'i').

RO4: Pay for vehicle fuel.

RO5: Return vehicle at date 'd'.

RO6: Pay for damages.

RO7: Settle all disputes.

RO8: Provide electronic evidences about breaches of the contract.

Renter's Rights

RR1: Terminate the x-contract, if the Owner is in breach of the contract.

Our next task is to map the above rights and obligations into FSMs for the Owner and the Renter.

5.2.3 The finite state machines

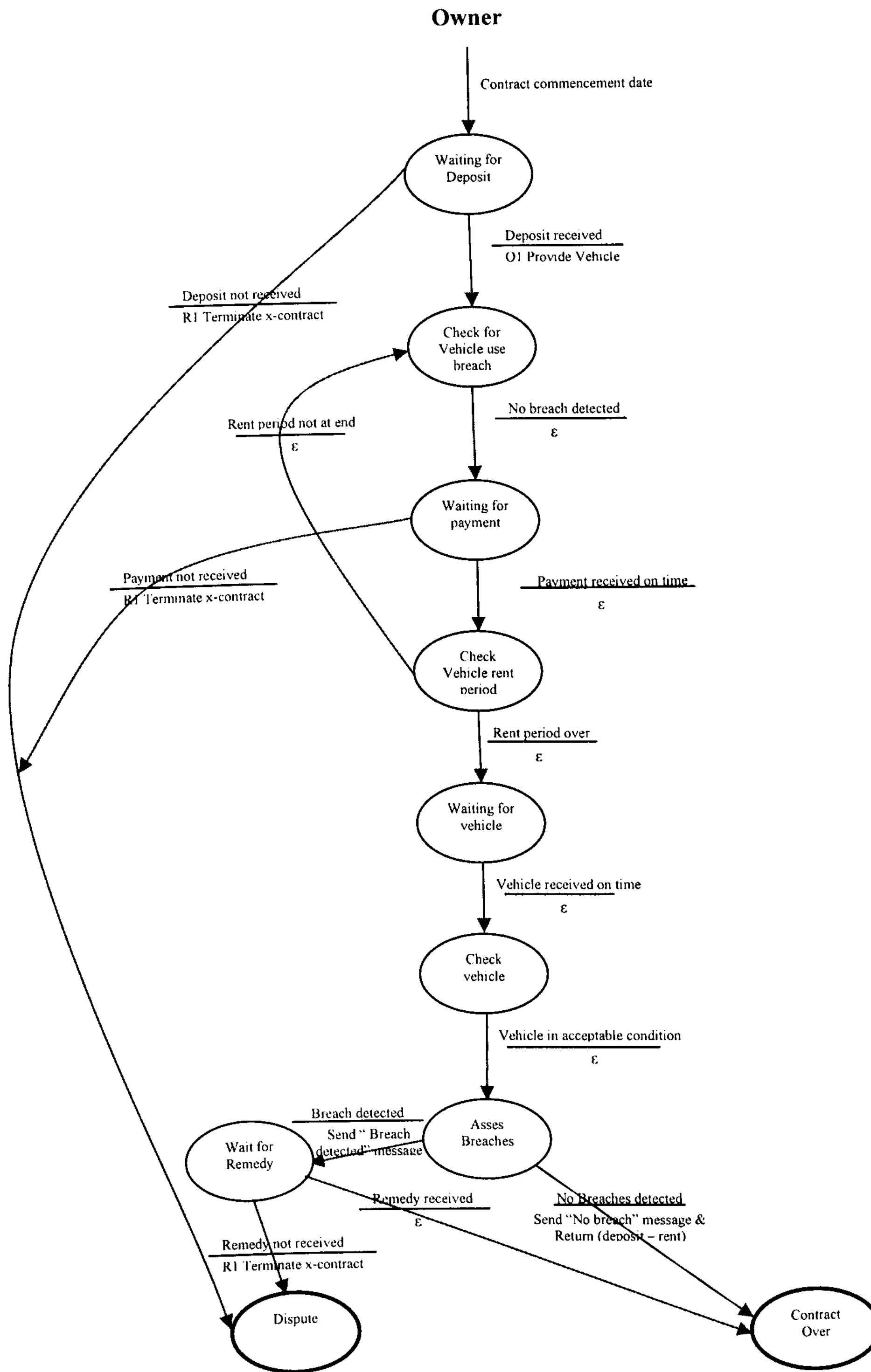


Fig.5.9. Owner's Finite state machines

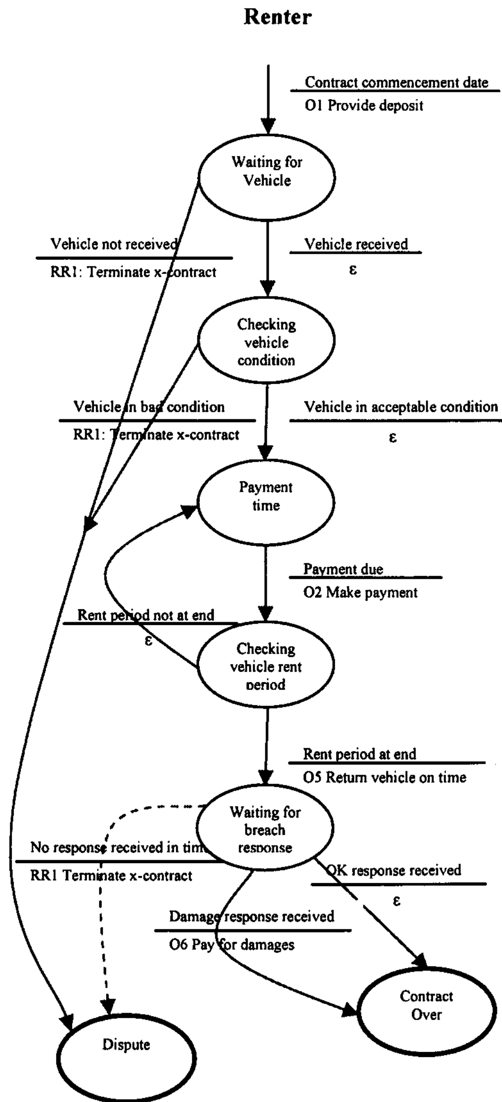


Fig.5.10. Renter's finite state machine

5.2.4 The Promela Model:

Below is the Promela verification model of the finite state machines presented in Section 5.2.3.

```
1   /*   Verification model of an x-contract between a car owner and
a renter
2     *
3     *   Programme created using XSPIN for simulation and
verification of FSM
4     *   correctness
5     *
6     * AUTHOR: Ellis Solaiman
7     * University Of Newcastle  Upon Tyne
8     * Date of Creation    May  10  2003
9     * Last Update         July 22  2003
10    *
11    */
12
13  // Constant definitions
14
15  #define Yes 1
16  #define No  0
17
18  #define Good 1
19  #define Bad 0
20
21  #define Accept 1
22  #define Reject 0
23
24  #define Over 1
25  #define NotOver 0
26
27  #define RentTime 5
28
29  //Messages that will be passed between Owner and the Renter
30  mtype = {Deposit, Payment, Response, Remedy, Breach}
31
32  chan R2O=[1] of {mtype, byte}; // channel Renter to Owner
33  chan O2R=[1] of {mtype, byte}; // channel Owner to Renter
34
35  byte provideVehicle = No;
36
37  /****** Owner FSM ******/
38
39  proctype Owner()
40  {
41  int  depositVal;
42  byte paymentGood;
43  byte vehicleBreach;
44  byte returnedInTime;
45  byte condition;
46  byte remedy = Good;
47  byte rentPeriod;
48  int time = 0;
49
50  /*** Waiting for deposit reciept ***/
51  WaitingforDeposit:
52
```

```
53  if
54  ::R2O ? Deposit(depositVal) -> //Deposit received
55  ::timeout -> goto Dispute //timeout assuption by the designer
56  fi;
57
58  // Random if/fi structure
59  if
60  :: provideVehicle = Yes; assert(depositVal==Good);
61  :: provideVehicle = No; goto end;
62  fi;
63
64  /*** Check for vehicle use breach ***/
65  CheckVehicleBreach:
66
67  if
68  :: vehicleBreach = Yes; goto WaitingforPayment;
69  :: vehicleBreach = No;  goto WaitingforPayment;
70  fi; // A breach or non-   breach is recorded and we carry on
71
72  /*** Waiting for rent payment ***/
73  WaitingforPayment:
74
75  if
76  ::R2O ? Payment(paymentGood) ->
77  ::timeout -> goto Dispute; // Assumption on the designers part
78  fi;
79
80  if
81  ::(paymentGood==Yes) -> goto RentPeriod;
82  ::(paymentGood==No) -> goto  RentPeriod;
83  fi; // Payment is recorded and we carry on
84
85  /*** Check vehicle rent period ***/
86  RentPeriod:
87
88  if
89  :: (time == RentTime); goto WaitingforVehicle;
90  :: (time != RentTime); time++; goto CheckVehicleBreach;
91  fi;
92
93  /*** Waiting for Vehicle to be returned ***/
94  WaitingforVehicle:
95
96  if
97  ::returnedInTime = Yes; goto CheckVehicle;
98  ::returnedInTime = No;  goto CheckVehicle;
99  fi;
100
101 /*** Check Vehicle for damage ***/
102 CheckVehicle:
103
104 if
105 ::condition = Good; goto AssessBreaches;
106 ::condition = Bad; goto AssessBreaches;
107 fi;
108
109 /*** Assess any breaches relating to rental of vehicle ***/
110 AssessBreaches:
111
112 if
```

```
113 :: (
(vehicleBreach==Yes)||(paymentGood==No)||(returnedInTime==No)||(condi
tion==Bad) ) ->
114    O2R ! Breach (Yes);  goto RemedyRequest;
115 :: else -> O2R ! Breach(No); goto ContractOver
116 fi;
117
118 /*** Breach detected and remedy requested ***/
119 RemedyRequest:
120
121 if
122 ::R2O ? Remedy(remedy) ->
123 ::timeout -> goto Dispute
124 fi;
125
126 // Check remedy correctness
127 if
128 :: remedy == Good -> goto ContractOver
129 :: remedy == Bad -> goto Dispute
130 fi;
131
132 /*** Dispute state. Owner not happy about something ***/
133 Dispute:
134
135 printf("\n\nDispute\n\n");
136 goto end;
137
138 /*** Contract ends satisfactorily ***/
139 ContractOver:
140 assert(remedy == Good);
141 printf("\n\nContract Over\n\n");
142
143 /*** End ***/
144 end:
145 printf("\n\n End \n\n");
146
147 }
148
149
150
151
152
153 /****** Renter FSM ******/
154 proctype Renter()
155 {
156
157 byte paymentVal;
158 byte rentPeriod;
159 byte response;
160 byte depositVal;
161 byte condition;
162 byte paymentDue;
163 int time = 0;
164
165
166 /*** Provide Deposit ***/
167 if
168 :: R2O ! Deposit(depositVal) ->
169 :: skip; goto End; // Modelling possible system failure
170 fi;                // or failure by the Renter to make the
deposit
```

```
171
172 /*** Waiting for Vehicle ***/
173 WaitforVehicle:
174
175 if
176 :: (provideVehicle == Yes); goto CheckVehicleCondition;
177 :: (provideVehicle == No); goto Dispute;
178 fi;
179
180 /***Check the vehicle condition***/
181 CheckVehicleCondition:
182
183 if
184 ::condition = Good -> goto PaymentTime;
185 ::condition = Bad -> goto Dispute;
186 fi;
187
188 /*** If it is due, pay the car rent ***/
189 PaymentTime:
190
191 if
192 :: paymentDue = Yes ->
193 :: paymentDue = No -> goto PaymentTime;
194 fi;
195
196 if
197 :: paymentVal = Good;
198 :: paymentVal = Bad;
199 fi;
200
201 if
202 :: R2O ! Payment(paymentVal) ->
203 :: skip; -> goto End // Renter does not pay for some reason.
204 fi;                  // Maybe he runs of with the car!
205
206 /***Check vehicle rent period***/
207 RentPeriod:
208
209 if
210 :: (time == RentTime) ->
211 :: (time != RentTime) -> time++; goto PaymentTime;
212 fi;
213
214 //Return vehicle
215 goto WaitingBreachResponse;
216
217
218 /***Waiting for any complaints from the Owner***/
219 WaitingBreachResponse:
220
221 if
222 ::O2R ? Breach(response) ->
223 ::timeout;
224 fi;
225
226 if
227 :: (response == Yes) ->
228 :: (response == No) -> goto ContractOver
229 fi;
230
231 //Car owner requests remedy
```

```
232
233 if
234 :: R2O ! Remedy (Good)-> goto ContractOver
235 :: R2O ! Remedy (Bad) -> goto ContractOver
236 fi;
237
238 /*** Contract Over ***/
239 ContractOver:
240 assert(condition == Good);
241 printf("\n\n Contract Over\n\n");
242 goto End;
243
244 /*** Dispute state. Renter not happy with something***/
245 Dispute:
246 printf("\n\n Dispute \n\n");
247
248 /*** End state ***/
249 End:
250 printf("\nEnd\n");
251
252 }
253
254 /****** Initiate Owner's and Renter's FSM ******/
255 init
256 {
257 run Owner();
258 run Renter();
259 }
```

5.2.5 X-contract verification

General Safety properties

After checking the Promela model for any syntax errors, we will first begin checking the correctness of the model with respect to the general correctness requirements.

Figure 5.11 shows the path in the verification model through which the Spin validator detected an ambiguity.

The reason for the sequence of events in the simulation is that throughout the model, we initially assumed that the Renter and the Owner may make mistakes in assessing whether the car rent period is over. So in figure 5.11, Spin simulates the possibility that the Renter assumes the rental period to be over, in the mean time the owner assumes that the rent period is not over - which one of the two assessments is correct does not matter for the purpose of validation-. This results in the Owner not receiving an expected payment, and he terminates the x-contract and goes to a dispute state. Meanwhile the Renter assuming there are no more rent payments to be made, returns the car, and moves on to the state waiting for the breach assessment from the owner, which will not be received because the Owner has already terminated the x-contract (Entering the dispute state), and is not sending any more messages within the confines of the x-contract. So we have deadlock because the Renter's FSM is waiting indefinitely, for a message which the Owner's FSM is not going to send. This could

be avoided for example by the Owner's FSM sending a payment reminder before terminating the x-contract.

```
Simulation Output
   1:   proc  0 (:init:) line 248 "pan_in" (state 1)    [(run Owner())]
   2:   proc  0 (:init:) line 249 "pan_in" (state 2)    [(run Renter())]
   3:   proc  2 (Renter) line 163 "pan_in" (state -)    [values: 1!Deposit,0]
   3:   proc  2 (Renter) line 163 "pan_in" (state 1)    [R2O!Deposit,depositVal]
   4:   proc  1 (Owner) line  51 "pan_in" (state -)     [values: 1?Deposit,0]
   4:   proc  1 (Owner) line  51 "pan_in" (state 1)     [R2O?Deposit,depositVal]
   5:   proc  1 (Owner) line  57 "pan_in" (state 6)     [provideVehicle = 1]
   6:   proc  1 (Owner) line  65 "pan_in" (state 11)    [vehicleBreach = 1]
   7:   proc  2 (Renter) line 171 "pan_in" (state 6)    [((provideVehicle==1))]
   8:   proc  2 (Renter) line 179 "pan_in" (state 12)   [condition = 1]
   9:   proc  2 (Renter) line 187 "pan_in" (state 18)   [paymentDue = 1]
  10:   proc  2 (Renter) line 192 "pan_in" (state 23)   [paymentVal = 1]
  11:   proc  2 (Renter) line 197 "pan_in" (state -)    [values: 1!Payment,1]
  11:   proc  2 (Renter) line 197 "pan_in" (state 27)   [R2O!Payment,paymentVal]
  12:   proc  2 (Renter) line 205 "pan_in" (state 32)   [rentPeriod = 1]
  13:   proc  1 (Owner) line  73 "pan_in" (state -)     [values: 1?Payment,1]
  13:   proc  1 (Owner) line  73 "pan_in" (state 17)
[R2O?Payment,paymentGood]
  14:   proc  1 (Owner) line  78 "pan_in" (state 22)    [((paymentGood==1))]
  15:   proc  1 (Owner) line  87 "pan_in" (state 30)    [rentPeriod = 0]
  16:   proc  1 (Owner) line  65 "pan_in" (state 11)    [vehicleBreach = 1]
  17:   proc  1 (Owner) line  74 "pan_in" (state 18)    [(timeout)]
Dispute
  18:   proc  1 (Owner) line 132 "pan_in" (state 65)
[printf('\\n\\nDispute\\n\\n')]
 End
  19:   proc  1 (Owner) line 141 "pan_in" (state 68)    [printf('\\n\\n End
\\n\\n')]
spin: trail ends after 20 steps
#processes: 3
  20:   proc  2 (Renter) line 216 "pan_in" (state 38)
  20:   proc  1 (Owner) line 143 "pan_in" (state 69)
  20:   proc  0 (:init:) line 250 "pan_in" (state 3)
3 processes created
Exit-Status 0
Single Step   Suspend   Save in:   Clear   Cancel
```

Fig .5.11 Safety error in the verification model

However this is not required within the text of the contract, and is not requested by the signatories, and therefore we have not coded a payment reminder into the x-contract model. The signatories have agreed instead to solve the deadlock possibility by assuming time conflicts between them regarding the rental period is not required, so this assumption is removed from the code by synchronising both their finite state machines using a timer, see lines 27, 89, 90, 209, and 210.

Even after synchronising both FSMs there may still arise the possibility that the Renter will wait endlessly for a breach response, so the signatories also agreed to give the Owner a time limit within which to send a breach complaint. If the breach complaint is not received within this time limit, the Renter himself goes to a dispute state. This is achieved by adding line 222, which is enacted if the Renter does not receive a response in line 221:

```
218 WaitingBreachResponse:
219
220 if
221 ::O2R ? Breach(response) ->
222 ::timeout -> goto Dispute;
223 fi;
224
```

As a result of this, the initial contract as well as the x-contract model is amended by adding a clause to remove this ambiguity:

Remedy

The Owner hereby agrees that he shall notify the Renter of any Remedy requests within a period after the return of the vehicle that does not exceed ______ number of agreed time units.

Also the Renter's finite state machine is modified, see dashed arrow in figure (Renter FSM).

After making the changes, we once again run the modified Promela model through the Spin verifier against the general safety requirements, and it does not detect any errors this time. See figure 5.12.

```
Verification Output
(Spin Version 4.0.1 -- 7 January 2003)
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (not selected)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid endstates       +

State-vector 60 byte, depth reached 67, errors: 0
     886 states, stored
     274 states, matched
    1160 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

2.622   memory usage (Mbyte)

unreached in proctype Owner
        (0 of 70 states)
unreached in proctype Renter
        (0 of 58 states)
unreached in proctype :init:
        (0 of 3 states)

Save in: [      ]  Clear  Close
```

Fig.5.12. Verification output after checking general safety requirements

Specific Safety properties

After ensuring the correctness of the model with respect to the general safety properties, we can begin checking it against any specific safety requirements that the signatories would like to test the model for. The parties have expressed the following requirements from the model:

1. The Owner would like to ensure that the x-contract will not allow the system go into a satisfactory situation before it has checked that the renter has remedied any breaches of the contract.

2. The Owner would like to ensure that the vehicle is not provided before a correct deposit is received.

2. The Renter would like to ensure that he will not receive a vehicle in a bad condition.

As we did in Section 5.1.5, we will check for these requirements by inserting assert statements in the relevant points of the Promela model:

For the first safety requirement:

```
138 /*** Contract ends satisfactorily ***/
139 ContractOver:
140 assert(remedy == Good);
141 printf("\n\nContract Over\n\n");
```

For the second safety requirmnent:

```
58  // Random if/fi structure
59  if
60  :: provideVehicle = Yes; assert(depositVal==Good);
61  :: provideVehicle = No; goto end;
62  fi;
```

For the third safety requirement:

```
238 /*** Contract Over ***/
239 ContractOver:
240 assert(condition == Good);
241 printf("\n\n Contract Over\n\n");
242 goto End;
```

Next, we setup the validator to check for assertions, and it detects a violation (assertion violation (depositVal == 1)). Through the route in which the violation was detected, figure 5.13, we realise that the Promela model does not check the deposit when it is received. This is because the text contract does not actually specify this as a requirement so the model designer did not specify the requirement within the x-contract model either.

```
Simulation Output
  1:    proc  0 (:init:) line 257 "pan_in" (state 1)    [(run Owner())]
  2:    proc  0 (:init:) line 258 "pan_in" (state 2)    [(run Renter())]
  3:    proc  2 (Renter) line 168 "pan_in" (state -)    [values: 1!Deposit,0]
  3:    proc  2 (Renter) line 168 "pan_in" (state 1)    [R20!Deposit,depositVal]
  4:    proc  2 (Renter) line 177 "pan_in" (state 8)    [((provideVehicle==0))]
 Dispute
  5:    proc  2 (Renter) line 246 "pan_in" (state 57)   [printf('\\n\\n Dispute \\n\\n')]
End
  6:    proc  2 (Renter) line 250 "pan_in" (state 58)   [printf('\\nEnd\\n')]
  7: proc 2 terminates
  8:    proc  1 (Owner) line  54 "pan_in" (state -)     [values: 1?Deposit,0]
  8:    proc  1 (Owner) line  54 "pan_in" (state 1)     [R20?Deposit,depositVal]
  9:    proc  1 (Owner) line  60 "pan_in" (state 6)     [provideVehicle = 1]
spin: line  60 "pan_in", Error: assertion violated
spin: text of failed assertion: assert((depositVal==1))
#processes: 2
 10:    proc  1 (Owner) line  60 "pan_in" (state 7)
 10:    proc  0 (:init:) line 259 "pan_in" (state 3)
3 processes created
Exit-Status 0
Single Step   Suspend        Save in: sim.out   Clear   Cancel
```

Fig.5.13. Simulation output through path where safety violation is detected

As the signatories have specified a correct deposit as a requirement, the Promela model is subsequently modified:

```
50  /*** Waiting for deposit reciept ***/
51  WaitingforDeposit:
52
53  if
54  ::R2O ? Deposit(depositVal) -> //Deposit received
55  ::timeout -> goto Dispute //timeout assuption by the designer
56  fi;
57
58  if
59  ::(depositVal == Good) ->
60  ::(depositVal == Bad) -> goto end
61  fi;
```

As can be seen in line 60, it was decided that if an incorrect deposit is received, then the contract would simply end, without the need to go to a dispute state. It was also decided that no changes to the original text contract are required.

Liveness properties

The parties have requested to test the x-contract model for the following liveness requirements:

1. The Renter makes correct Rent payments at the required time, otherwise the Owner's FSM does not proceed, and goes to a dispute state.
2. The x-contract does not get into a situation where the Renter must make rent payments infinitely.

Liveness requirement 1 can be defined as of type CR10 (Responsiveness). We will test for this by defining the variable *payInTime,* and asserting that *payInTime*, and *paymentGood*, are always True when in the "Check vehicle rent period" state:

```
91  /*** Check vehicle rent period ***/
92  RentPeriod:
93  assert(payInTime == Yes && paymentGood == Yes);
```

The validator detects no error, therefore requirement 1 is valid.
Requirement 2 tests for livelock (CR 9). We want to test that neither of the FSMs loops infinitely through a sequence of steps that involves the Renter having to infinitely pay (monthly for example) rent for the car, meaning that a rented car cannot be rented for ever.

To test for this requirement, we will insert an *accept* labels in both FSMs as follows:

```
79  WaitingforPayment:
80  acceptNotWaitPay:

195 PaymentTime:
196 acceptNotPay:
```

Inserting the accept labels after the `WaitingforPayment`, and the `PaymentTime` statesSpin states tells the verifier to detect whether the x-contract model allows a hypothetical scenario where any of these two states could be executed infinitely. We next set the Basic verification options of the Spin validator to detect livelocks. See figure (Livelock).

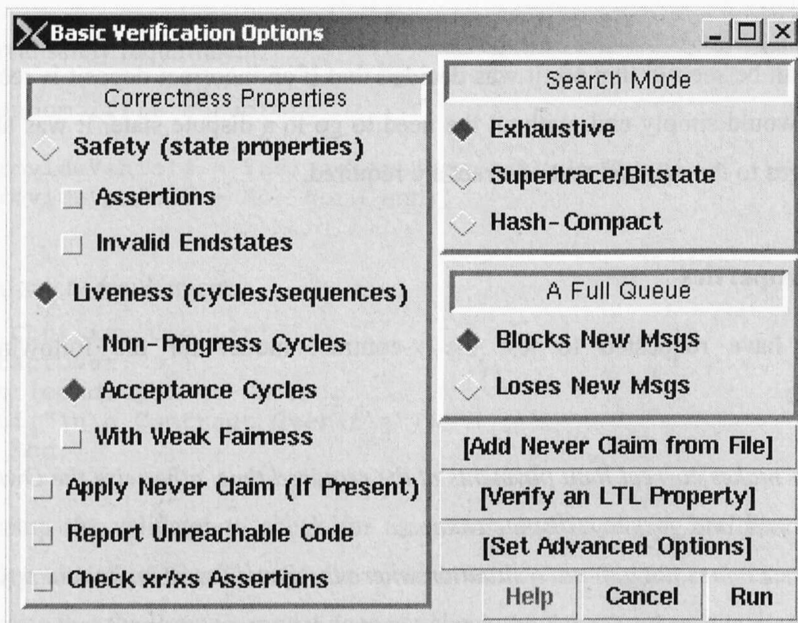


Fig.5.14. Verification options set to detect livelock

We run the verifier, and no violations are detected. This is because we had inserted a RentTime limit into the model:

```
91   /*** Check vehicle rent period ***/
92   RentPeriod:
93   assert(payInTime == Yes && paymentGood == Yes);
94   if
95   :: (time == RentTime); goto WaitingforVehicle;
96   :: (time != RentTime); time++; goto CheckVehicleBreach;
97   fi;

212 /***Check vehicle rent period***/
213 RentPeriod:
214
215 if
216 :: (time == RentTime) ->
```

```
217 :: (time != RentTime) -> time++; goto PaymentTime;
218 fi;
```

After verifying the correctness of the model with respect to the safety and liveness requirements, we can proceed with implementing the x-contract.

5.3 Playing a game over a network

Turn based Games played between two or more players over a network, can serve as good examples of applications which contain rules that govern the interaction between parties. Applications with such rules require validation for correctness before they can be implemented. Examples of games that can be played remotely are; Chess, Ticktacktoe, Monopoly, Poker…etc.

Two or more players before starting a game will be familiar with the set of conditions under which the game must be played. Conditions or rules can be seen as the "contract clauses" that the players must agree to before beginning.

Games vary in the number of players allowed to participate from a minimum of two, going anything up to 6 or more. Generally online games, have a familiar pattern of communication based on each player waiting for their turn to arrive, making judgements based on the game status, and eventually doing something. Throughout the game the players will observe other player's actions to ensure that everyone attends to the game conditions.

In this section, we present a possible finite state machine representation of some turn based games played over a network. The finite state machine should be detailed and unambiguous to facilitate the interaction between the players in an orderly and lawful manner. We will begin therefore by formulating the general set of rules or "contract clauses" that the players must adhere to during game play.

A helpful reminder would be that all applications from business contracts to games are to be implemented in a decentralised distributed fashion, with the help of a suitable middleware service such as B2BObjects. Therefore the traditional method of running games from a dedicated central server is not applicable for our implementation. Each of the participants in the game will have a copy of the state of the game as well as the rules of the game in the form of identical object copies. Every player will attempt to make changes to the state of the game by interfacing with his/her local object copy, and transmitting the attempted changes to the other player's object copies. Only when all players accept the attempted change to the game status, will every player's object copy be modified.

5.3.1 Rules of the game

This is a game of "chance" between "3 players" to be known as: "Player1", "Player2", and "Player3". Following the beginning of the game, the players will adhere to the following rules:

```
1. A player p must make an "action" within 2 minutes of receiving the "turn" or they will be declared "defeated".
2. The turn will not be with more than one player at any one time.
3. The player with the "turn" must send his/her chosen action to all players.
4. Upon receipt of an action from the "turn" player, the receiving player must send an "action accepted" or "action rejected" message to the turn player.
5. If the turn player receives an "action rejected" message from the other players then he/she must make another action within 2 minutes of receiving the rejecting message.
6. If the turn player p receives an "action accepted" message from the other players, then the turn player must send the "turn" to the next player p+1.
7. If the turn player sends an action that is judged by the other players to be a "winning action" then the game ends, and the turn player that performed the action is declared winner of the game.
8. A player may resign from the game only when he/she has the turn.
```

5.3.2 Players' rights and obligations

There are two possible methods that we can choose from in order to convert the rules in Section 5.3.1 into an x-contract. We could either split the contract into rights and obligations for the turn player, and rights and obligations for the non-turn players, so each player will have two finite state machines that his/her system switches between based on whether the player has the turn or not. Or we can simply extract rights and obligations for each player that represent the game rules as a whole, so each player will only have one FSM. To demonstrate the turn being passed between the players, we have chosen to implement the second approach.

Players' Obligations

PO1: A player must make an "action" within 2 minutes of receiving the "turn".

PO2: Turn player must send chosen action to all players.

PO3: Non-turn players must send Action response to "turn" player.

PO4: Turn player must make an action within 2 minutes of receiving a rejection message.

PO5: Turn player must send the turn to the next player after receiving an action accept message from all the other players.

Players' Rights

PR1: Resign from the game (only when the player has the turn)

5.3.3 The finite state machine

Figure 5.14 shows a FSM which models the general game interaction pattern. The figure shows the FSM for just one player. The FSM will be the same for all the players.

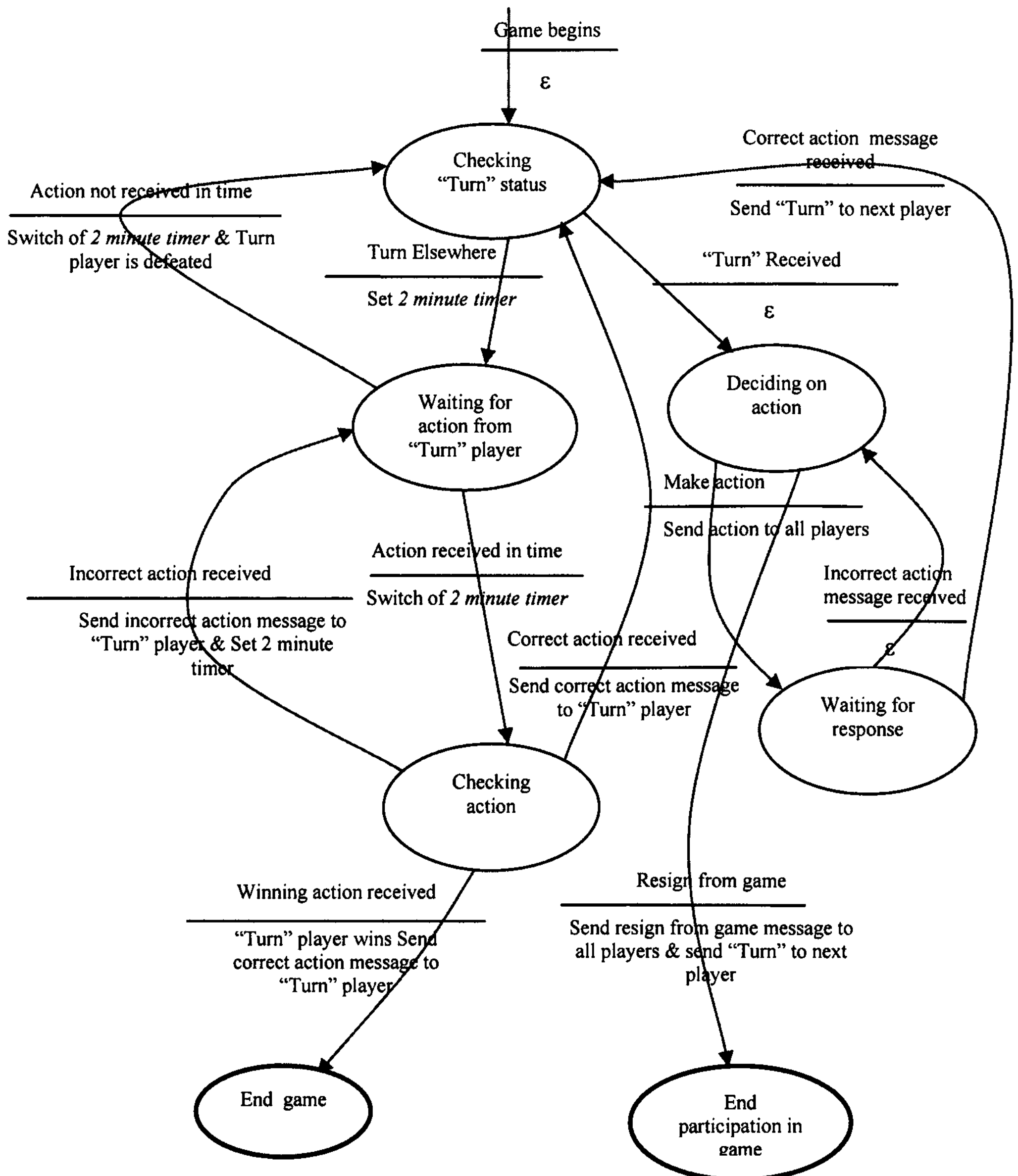


Fig. 5.14. FSM for a player participating in a turn based game

5.3.4 Games' FSMs in Promela

We will simulate a game between 3 players. As the finite state machine for each player will be the same, the Promela model will have virtually identical procedures for each of the players:

```
1   /*   Verification model of a game played between 3 players
2    *   Programme created using XSPIN for simulation and
verification of FSM
3    *   correctness
4    *
5    * AUTHOR: Ellis Solaiman
6    * University Of Newcastle  Upon Tyne
7    * Date of Creation    May  10  2003
8    * Last Update         July 24  2003
9    *
10   */
11
12  #define ON 1
13  #define OFF 0
14
15  #define Good 1
16  #define Bad 0
17
18  #define Yes 1
19  #define No 0
20
21  #define True 1
22  #define False 0
23
24  #define Win 10
25
26  #define Resign 5
27
28  mtype = {Action, ActionA, ActionB, ActionC, Turn, Response,
ResponseA, ResponseB, ResponseC}
29
30  chan P2P[4] = [1] of {mtype, int};
31
32  int turn = 1; // Turn begins with first player
33  byte player[4];
34  byte turnDecided = Yes;
35
36
37
38
39
40
41
42  /****** FSM of first player ******/
43  proctype PlayerA()
44  {
45  int id = 1;
46  player[1] = ON;
47
48  int action;
49  int response[3];//2 players respond
```

```
50  response[0] = 0;
51
52
53  /*** Is it my turn? ***/
54  CheckingTurnStatus:
55
56  do // Loop untill the turn has been decided.
57  :: (turnDecided == Yes) ->
58      if
59      :: (turn == id) -> goto DecidingOnAction;
60      :: (turn != id) -> goto WaitingForAction;
61      fi; break;
62  :: (turnDecided == No);
63  od;
64
65
66
67  /*** It is my turn. Deciding on action ***/
68  DecidingOnAction:
69
70  turnDecided = No;
71  //Randomly simulate one of the following actions
72  if
73  :: action = Good;
74  :: action = Bad;
75  :: action = Win;
76  :: action = Resign -> goto EndParticipation
77  fi;
78
79  if //If other players have left the game, then end the game.
80  :: (player[2] != ON && player[3]!= ON) -> goto EndParticipation;
81  :: else ->
82  fi;
83
84  // Randomly send action or end participation in game
85  if
86  :: if
87     :: (player[2] == ON) -> P2P[id] ! ActionB(action);
88     :: else ->
89     fi;
90
91     if
92     :: (player[3] == ON) -> P2P[id] ! ActionC(action);
93     :: else ->
94     fi;
95  ::skip -> goto EndParticipation;
96  fi;
97
98
99  //Waiting for response to action from other participants
100 WaitForResponse:
101
102 if //First check that the player (P2) I am waiting a response
from is playing
103 :: (player[2] == ON) -> P2P[2] ? ResponseB(response[1]) ->
104 :: else ->
105 fi;
106
107 if //Check that the player (P3) I am waiting a response from is
playing
108 :: (player[3] == ON) -> P2P[3] ? ResponseC(response[2]) ->
```

```
109 :: else ->
110 fi;
111
112 if
113 :: ((player[2] == ON) && (response[1] == Bad)) -> goto
DecidingOnAction;
114 :: ((player[3] == ON) && (response[2] == Bad)) -> goto
DecidingOnAction;
115 :: else ->
116 fi;
117
118 if // if accepted action was a winning one
119 :: (action == Win) -> goto End
120 :: else ->
121 fi;
122
123 if // If we reach here then the move was judged by the players to
be good
124 :: (player[2] == ON) ->  turn = 2; turnDecided = Yes; goto
CheckingTurnStatus
125 :: else if
126           :: (player[3] == ON) -> turn = 3; turnDecided = Yes;
goto CheckingTurnStatus
127           :: else -> printf("\n\n Unexpected Event\n\n");
128          fi;
129 fi;
130
131
132
133
134 /*** It is not my turn, Waiting for action from player with turn
***/
135 WaitingForAction:
136
137 if
138 :: (player[turn] == ON) ->
139       if
140       :: P2P[turn] ? ActionA(action) ->
141       :: timeout -> player[turn] = OFF ->
142     if
143     :: (turn == 2) -> turn=3; turnDecided = Yes; goto
CheckingTurnStatus
144     :: (turn == 3) -> turn=1; turnDecided = Yes; goto
CheckingTurnStatus
145     fi;
146       fi;
147 :: else -> goto CheckingTurnStatus;
148 fi;
149
150 CheckingAction:
151
152 if
153 ::(action == Good) -> P2P[id] ! ResponseA(Good) -> goto
CheckingTurnStatus
154 ::(action == Win) -> P2P[id] ! ResponseA(Good) -> goto End
155 ::(action == Bad) -> P2P[id] ! ResponseA(Bad) -> goto
WaitingForAction
156 fi;
157
158 EndParticipation:
159 printf("\n\nPlayerA Resigned\n\n");
```

```
160
161 End:
162 printf("\n\nPlayerA Game Ended\n\n");
163
164 }
165
166
167
168
169
170
171
172 /****** FSM of second player ******/
173 proctype PlayerB()
174 {
175 int id = 2;
176 player[2] = ON;
177
178 int action;
179 int response[3];//2 players respond
180 response[0] = 0;
181
182 /*** Is it my turn? ***/
183 CheckingTurnStatus:
184
185 do // Loop untill the turn has been decided.
186 :: (turnDecided == Yes) ->
187     if
188     :: (turn == id) -> goto DecidingOnAction;
189     :: (turn != id) -> goto WaitingForAction;
190     fi; break;
191 :: (turnDecided == No)
192 od;
193
194
195 /*** It is my turn. Deciding on action ***/
196 DecidingOnAction:
197
198 turnDecided = No;
199 //Randomly simulate one of the following actions
200 if
201 :: action = Good;
202 :: action = Bad;
203 :: action = Win;
204 :: action = Resign -> goto EndParticipation
205 fi;
206
207
208 if //If other players have left the game, then end the game.
209 :: (player[1] != ON && player[3]!= ON) -> goto EndParticipation;
210 :: else ->
211 fi;
212
213 // Randomly send action or end participation in game
214 if
215 :: if
216    :: (player[1] == ON) -> P2P[id] ! ActionA(action);
217    :: else ->
218    fi;
219
220    if
```

```
221     :: (player[3] == ON) -> P2P[id] ! ActionC(action);
222     :: else ->
223     fi;
224 ::skip -> goto EndParticipation;
225 fi;
226
227
228 //Waiting for response to action from other participants
229 WaitForResponse:
230
231 if //First check that the player (P1) I am waiting a response
from is playing
232 :: (player[1] == ON) -> P2P[1] ? ResponseA(response[1]) ->
233 :: else ->
234 fi;
235
236 if //Check that the player (P3) I am waiting a response from is
playing
237 :: (player[3] == ON) -> P2P[3] ? ResponseC(response[2]) ->
238 :: else ->
239 fi;
240
241
242 if
243 :: ((player[1] == ON) && (response[1] == Bad)) -> goto
DecidingOnAction;
244 :: ((player[3] == ON) && (response[2] == Bad)) -> goto
DecidingOnAction;
245 :: else ->
246 fi;
247
248
249 if // if accepted action was a winning one
250 :: (action == Win) -> goto End
251 :: else ->
252 fi;
253
254
255 if // If we reach here then the move was judged by the players to
be good
256 :: (player[3] == ON) ->  turn = 3; turnDecided = Yes; goto
CheckingTurnStatus
257 :: else if
258           :: (player[1] == ON) -> turn = 1; turnDecided = Yes;
goto CheckingTurnStatus
259           :: else -> printf("\n\n Unexpected Event\n\n");
260          fi;
261 fi;
262
263
264
265 /*** It is not my turn, Waiting for action from player with turn
***/
266 WaitingForAction:
267
268 if
269 :: (player[turn] == ON) ->
270       if
271       :: P2P[turn] ? ActionB(action) ->
272       :: timeout -> player[turn] = OFF ->
273      if
```

```
274     :: (turn == 1) -> turn=2; turnDecided = Yes; goto
CheckingTurnStatus
275     :: (turn == 3) -> turn=1; turnDecided = Yes; goto
CheckingTurnStatus
276     fi;
277       fi;
278 :: else -> goto CheckingTurnStatus;
279 fi;
280
281
282 CheckingAction:
283
284 if
285 ::(action == Good) -> P2P[id] ! ResponseB(Good) -> goto
CheckingTurnStatus
286 ::(action == Win) -> P2P[id] ! ResponseB(Good) -> goto End
287 ::(action == Bad) -> P2P[id] ! ResponseB(Bad) -> goto
WaitingForAction
288 fi;
289
290
291
292
293
294 EndParticipation:
295 printf("\n\nPlayerB Resigned\n\n");
296
297 End:
298 printf("\n\nPlayerB Game Ended\n\n");
299
300 }
301
302
303
304
305 /****** FSM of third player ******/
306 proctype PlayerC()
307 {
308 int id = 3;
309 player[3] = ON;
310
311 int action;
312 int response[3];//2 players respond
313 response[0] = 0;
314
315 /*** Is it my turn? ***/
316 CheckingTurnStatus:
317
318 do // Loop untill the turn has been decided.
319 :: (turnDecided == Yes) ->
320     if
321     :: (turn == id) -> goto DecidingOnAction;
322     :: (turn != id) -> goto WaitingForAction;
323     fi;  break;
324 :: (turnDecided == No)
325 od;
326
327
328 /*** It is my turn. Deciding on action ***/
329 DecidingOnAction:
330
```

```
331 turnDecided = No;
332 //Randomly simulate one of the following actions
333 if
334 :: action = Good;
335 :: action = Bad;
336 :: action = Win;
337 :: action = Resign -> goto EndParticipation
338 fi;
339
340 if //If other players have left the game, then end the game.
341 :: (player[1] != ON && player[2]!= ON) -> goto EndParticipation;
342 :: else ->
343 fi;
344
345 // Randomly send action or end participation in game
346 if
347 :: if
348    :: (player[1] == ON) -> P2P[id] ! ActionA(action);
349    :: else ->
350    fi;
351
352    if
353    :: (player[2] == ON) -> P2P[id] ! ActionB(action);
354    :: else ->
355    fi;
356 ::skip -> goto EndParticipation;
357 fi;
358
359
360 //Waiting for response to action from other participants
361 WaitForResponse:
362
363 if //First check that the player (P1) I am waiting a response
from is playing
364 :: (player[1] == ON) -> P2P[1] ? ResponseA(response[1]) ->
365 :: else ->
366 fi;
367
368 if //Check that the player (P3) I am waiting a response from is
playing
369 :: (player[2] == ON) -> P2P[2] ? ResponseB(response[2]) ->
370 :: else ->
371 fi;
372
373
374 if
375 :: ((player[1] == ON) && (response[1] == Bad)) -> goto
DecidingOnAction;
376 :: ((player[2] == ON) && (response[2] == Bad)) -> goto
DecidingOnAction;
377 :: else ->
378 fi;
379
380
381 if // if accepted action was a winning one
382 :: (action == Win) -> goto End
383 :: else ->
384 fi;
385
386
```

```
387 if // If we reach here then the move was judged by the players to
be good
388 :: (player[1] == ON) ->  turn = 1; turnDecided = Yes; goto
CheckingTurnStatus
389 :: else if
390          :: (player[2] == ON) -> turn = 2; turnDecided = Yes;
goto CheckingTurnStatus
391          :: else -> printf("\n\n Unexpected Event\n\n");
392         fi;
393 fi;
394
395
396 /*** It is not my turn, Waiting for action from player with turn
***/
397 WaitingForAction:
398
399 if
400 :: (player[turn] == ON) ->
401       if
402       :: P2P[turn] ? ActionC(action) ->
403       :: timeout -> player[turn] = OFF ->
404     if
405     :: (turn == 1) -> turn=2; turnDecided = Yes; goto
CheckingTurnStatus
406     :: (turn == 2) -> turn=3; turnDecided = Yes; goto
CheckingTurnStatus
407     fi;
408       fi;
409 :: else -> goto CheckingTurnStatus;
410 fi;
411
412
413 CheckingAction:
414
415 if
416 ::(action == Good) -> P2P[id] ! ResponseC(Good) -> goto
CheckingTurnStatus
417 ::(action == Win) -> P2P[id] ! ResponseC(Good) -> goto End
418 ::(action == Bad) -> P2P[id] ! ResponseC(Bad) -> goto
WaitingForAction
419 fi;
420
421
422
423
424
425 EndParticipation:
426 printf("\n\nPlayerC Resigned\n\n");
427
428 End:
429 printf("\n\nPlayerC Game Ended\n\n");
430
431 }
432
433
434
435 init
436 {
437
438 run PlayerA();
439 run PlayerB();
```

```
440 run PlayerC();
441
442 }
```

5.3.5 Game model verification

Because this is a game of rules that we are imposing on players who wish to play it, we will suffice with checking that the model is correct with respect to the general safety properties, i.e., CR3: Reachability, CR4: Freedom from deadlocks, and CR11: Absence of unsolicited responses.

We set the verifier to detect general safety properties, and run it. The verifier signals that it has detected an error (figure 5.15).

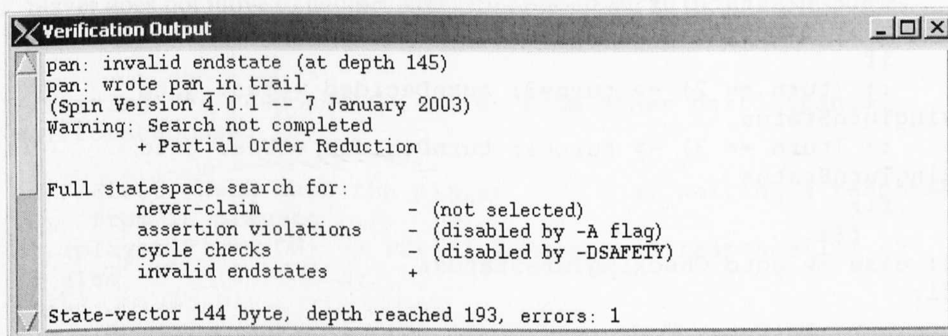


Fig.5.15. General safety error detected in Game model

To discover the source of the error, we will run the Spin simulator through the path in which the error was detected. The results are presented in figure 5.16.

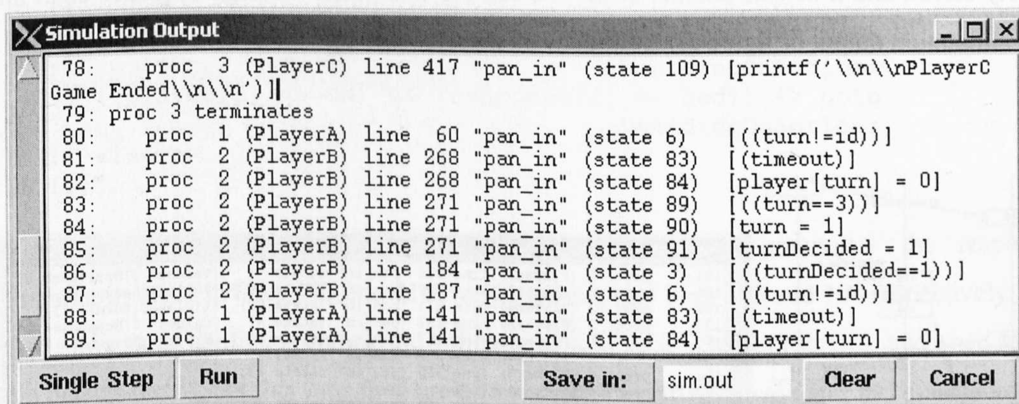


Fig.5.16. Simulation output of path in which error is detected

In the simulated route, the turn is passed to player C, who decides to abruptly leave the game without informing the other players. (steps 78, and 79 in figure 5.16).

Players A and B in the mean time are still waiting for an action from player C. Player B first detects that no action is forthcoming, and assigns the player with the turn player

C = OFF (step 82), and gives the turn (A global variable in our model) to player A (step 84). Following this, player A also detects that no action is forthcoming from player C. But because the global variable "turn" has been set by player B to equal "1", (i.e. turn = player A), player A unknowingly switches himself off! (step 89). This problem is an example of Spin's ability to detect programming mistakes, and can be corrected by both players checking if the player with the turn is "ON" before proceeding with waiting for an action:

```
136 /*** It is not my turn, Waiting for action from player with turn
***/
137 WaitingForAction:
138
139 if
140 :: (player[turn] == ON) ->
141        if
142        :: P2P[turn] ? ActionA(action) ->
143        :: timeout -> player[turn] = OFF ->
144     if
145     :: (turn == 2) -> turn=3; turnDecided = Yes; goto
CheckingTurnStatus
146     :: (turn == 3) -> turn=1; turnDecided = Yes; goto
CheckingTurnStatus
147     fi;
148        fi;
149 :: else -> goto CheckingTurnStatus;
150 fi;
```

Previously, line 140, and its counterparts in the FSMs of the other players did not exist.

After making this modification, we again run the Spin validator to check for general safety errors, and it detects another error. We can see this in the message sequence chart and the simulation output of figure 5.17.

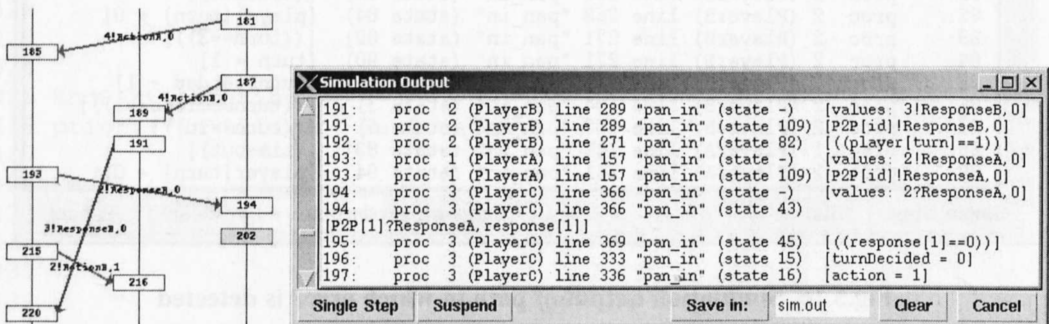


Fig.5.17. Simulation of second problem in game model

As can be seen at steps 181, and 187 of the message sequence chart in figure 5.17, when it is his turn, player C sends a bad action (knowingly or not knowingly) to players A, and B.

Player B at step 191, and Player A at step 193 of the simulation send player C an error message. However, Player A's response is received first, and player C goes back to the "Deciding on action state", and resigns from the game (step 202). Player B's response remains in a queue. It now becomes player A's turn. He makes his move, and waits for a response from player B, but instead receives player B's response to player C's move at step 187! This is an error caused by a mistake in the configuration of the message channels in the Promela code. In order to simplify the code, we setup 3 response channels, ResponseA, ResponseB, and ResponseC to be used respectively by each of the players in order that they can respond to the "turn" player's action. A more accurate modelling would be to setup channels as such: ResponseAB, ResponseAC, ResponseBA, ResponseBC ... etc. However this would needlessly complicate the model, and a better solution would be to ensure that the "turn" player does not move to another state before receiving all incoming responses:

```
99  //Waiting for response to action from other participants
100 WaitForResponse:
101
102 if //First check that the player (P2) I am waiting a response
         from is playing
103 :: (player[2] == ON) -> P2P[2] ? ResponseB(response[1]) ->
104 :: else ->
105 fi;
106
107 if //Check that the player (P3) I am waiting a response from is
        playing
108 :: (player[3] == ON) -> P2P[3] ? ResponseC(response[2]) ->
109 :: else ->
110 fi;
111
112 if
113 :: ((player[2] == ON) && (response[1] == Bad)) -> goto
                                                 DecidingOnAction;
114 :: ((player[3] == ON) && (response[2] == Bad)) -> goto
                                                 DecidingOnAction;
115 :: else ->
116 fi;
```

The above code is the corrected version, whereas previously the response comparisons made in lines 113 and 114, were placed under lines 103 and 108 respectively.

We must continue running the validator as long as it detects problems, and once again Spin detects errors, but this time they are of a different nature. The validator has detected code that is unreachable (CR3). See figure 5.18.

```
Verification Output
unreached in proctype PlayerA
        line 127, state 74, "printf('\n\n Unexpected Event\n\n')"
        (1 of 112 states)
unreached in proctype PlayerB
        line 259, state 74, "printf('\n\n Unexpected Event\n\n')"
        (1 of 112 states)
unreached in proctype PlayerC
        line 391, state 74, "printf('\n\n Unexpected Event\n\n')"
        (1 of 112 states)
unreached in proctype :init:
        (0 of 4 states)
```

Fig.5.18. Detection of unreachable code by the Spin validator.

Fortunately, this is code we inserted only to help us with the modelling process, and is not required for the game model or the implementation. We can therefore delete or comment out the code signalled as unreachable by the Spin verifier. This is a good example of Spin's ability to detect code and states that may not have been integrated correctly during the coding of a FSM. We comment out the unreachable code (lines 127, 259, and 391), run the verifier one more time for general safety properties, and it signals that the model is correct for these properties.

As mentioned earlier, we are not required to check the model for other correctness properties. The FSM model can now be implemented.

5.4 Summary

The focus of Chapter 4 and of this Chapter has been on the identification of a process that we can implement in order to test contracts for errors and ambiguities, and in order to remove these errors and ambiguities if any are detected.

We have identified a set of safety and liveness contract correctness requirements, and using these we have shown through examples and with the Spin model checker how inconsistencies can be detected and removed from contracts. Once a contract has been freed of errors, our next task is to create and implement the x-contract. This is the topic of Chapter 6.

Chapter Six

Middleware Support for X-Contract Implementation

In this chapter, we investigate what middleware services are required to support a contract management system that guarantees that the rights and obligations stipulated in a contract are monitored and enforced. We are assuming that the organizations involved might not trust each other, so an important requirement from the middleware which will facilitate the contractual interactions between the parties is that it should enable regulated transactions (as encoded in the x-contract) between two or more mutually suspicious but autonomous organizations.

It is clearly not possible to prevent organisations from misbehaving and attempting to cheat on their agreed contractual relationships. The best that can be achieved is to ensure that all contractual interactions between such organisations are funnelled through (a centralised or distributed) contract management system and that either (a) all other non-contractual interactions are disallowed, or (b) the contract management system is at least capable of monitoring and signalling the signatories of the contract as to when the contract is being violated, or ideally (c) both a, and b.

The *safety* properties of the x-contract implementation must ensure that local policies of an organization are not compromised despite failures and/or misbehavior by other parties; whilst the *liveness* properties should ensure that if all the parties are correct (not misbehaving), then agreed interactions would take place despite a bounded number of temporary network and computer related failures. Also because we are dealing with contracts, for the purposes of proof and legality the middleware must have means for collecting non-repudiable evidence of the actions of parties that interact with each other.

For non-repudiable information sharing we propose to use the B2BObject middleware developed at the University of Newcastle upon Tyne [CSW02].

6.1 Overview of B2BObjects middleware

B2Bobjects middleware service collects non-repudiable evidence for information sharing between parties that do not necessarily trust each other. Once deployed, each party holds a local copy of shared information encapsulated in objects. Access to and update of this information is subject to non-repudiable validation by each party. It is assumed that each organization has a local set of policies for information sharing that is consistent with the

overall information sharing agreement between the organizations (this agreement will be encoded in the x-contract). B2BObjects provides for the safety and liveness properties discussed at the beginning of this chapter. The safety property of B2BObjects ensures that local policies of an organization are not compromised despite failures and/or misbehaviour by other parties; whilst the liveness property ensures that if all the parties are performing their actions correctly as stipulated within a contract, then agreed interactions would take place despite a bounded number of temporary network and computer related failures.

Essentially, B2BObjects resembles a transactional object replica management system where each organization has a local copy of the object(s) to be shared. Any local updates to the copy by an organization ("proposed state changes" by the organization) are propagated to all the other organizations holding copies in order for them to perform local validation; a proposal comprises the new state and the proposer's signature on that state. Each recipient produces a response comprising a signed receipt and a signed decision on the (local) validity of the state change. All parties receive each response and a new state is valid if the collective decision is unanimous agreement to the change. The signing of evidence generated during state validation binds the evidence to the relevant key-holder. Evidence is stored systematically in local non-repudiation logs. For protocol details, see [CSW02].

State changes are subject to a locally evaluated validation process. State validation is application-specific and may be arbitrarily complex (and may involve back-end processes at each organisation).

Figure 6.1, presents four enterprises (E1, E2, E3, E4), sharing a state through three B2BObjects (A, B, and C). As shown in the figure, the logical view of shared objects in a virtual space (a) is realised by the regulated coordination of actions on object replicas held at each organisation (b).

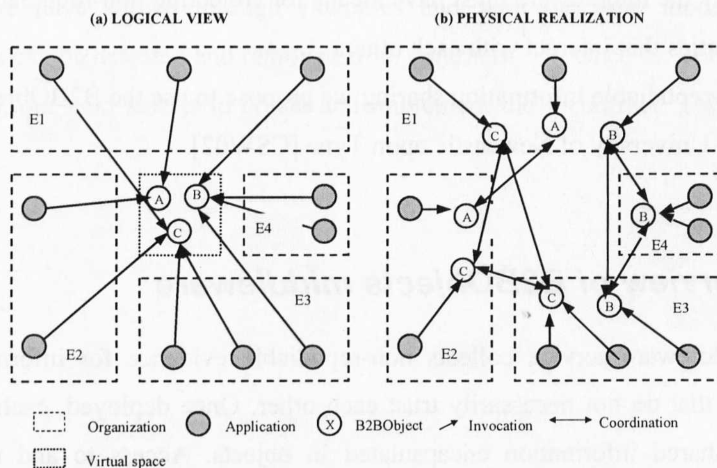


Fig.6.1. B2Bobject Interactions

6.2 B2BObjects API

The primary B2BObjects API classes are *B2BObject* — the application-specific augmentation of a local object, and *B2BObjectController* — the local interface to configuration, initiation, and control of information sharing. The interfaces to these classes and the relationship between them and the *B2BCoordinator* package are shown in Figure 6.2. The coordinator package manages inter-organisational connection to and communication between objects, and implements coordination protocols. It also provides state checkpointing, certificate management and non-repudiation services.

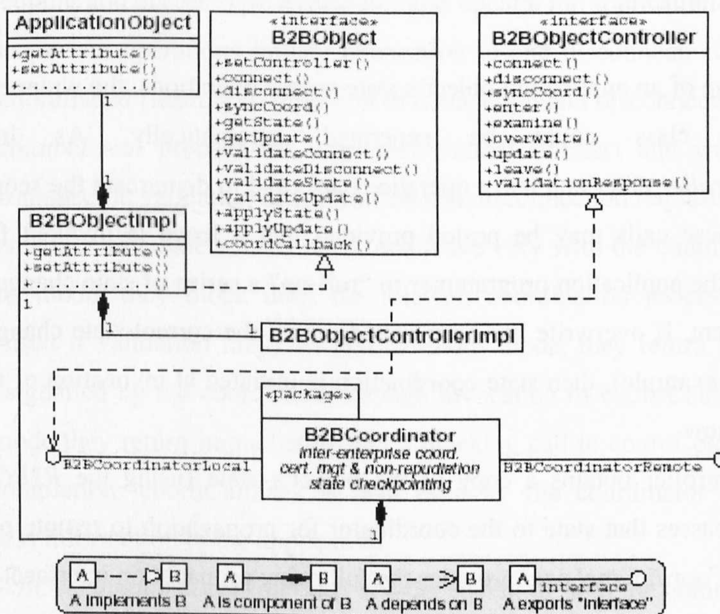


Fig.6.2. B2BObjects API

The *B2BObject* interface is implemented by the application programmer. The programmer decides whether to produce a new application object that implements both the *B2BObject* interface and the application logic, or to extend an existing application object, or to wrap the object with an implementation of the *B2BObject* interface. For example, the *ApplicationObject* operation:

```
setAttribute(AType a);
```

shown in figure 6.2, has a corresponding *B2BObjectImpl* wrapper operation that could be implemented as follows:

```
setAttribute(AType a) {
// start of state access
controller.enter();
// indicate overwriting object state
```

```
controller.overwrite();
// set the attribute
appObject.setAttribute(a);
// end of state access
controller.leave();
}
```

Similarly, the *B2BObjectImpl* `getAttribute` wrapper is:

```
AType getAttribute() {
// start of state access
controller.enter();
// indicate reading object state
controller.examine();
// get the attribute
AType attr = appObject.getAttribute();
// end of state access
controller.leave();
return attr;
}
```

Given knowledge of an application object's state access operations, the wrapper methods of a *B2BObjectImpl* class could be generated automatically. As indicated, the *B2BObjectController* enter and leave operations are used to demarcate the scope of access to object state. These calls may be nested provided that a leave is invoked for each enter. Nesting allows the application programmer to "roll-up" a series of state changes into a single coordination event. If overwrite has been called within the current state change scope (as in the setAttribute example), then state coordination is initiated at invocation of the final leave, as we describe now.

The controller obtains a copy of the object's state (using the *B2BObject* getState operation) and passes that state to the coordinator for propagation to remote parties for state validation. *B2BCoordinatorLocal* provides the following propagation interface:

```
public interface B2BCoordinatorLocal {
public void propagateConnect(String
coordAlias);
public void propagateDisconnect(String
subjectAlias);
public void propagateNewState(
NewStateRequest stateRequest);
...
...
}
```

A call to propagateNewState results in state validation at the remote parties via invocation of validateState on their copy of the shared object. The *B2BObjectController* validationResponse communicates the result of this application-specific validation. It can be invoked synchronously or asynchronously as a callback on the local controller. If a proposed change is accepted by all parties, an applyState call on each replica installs the newly validated state. Thus the leave operation implicitly invokes the state coordination protocol, via the local coordinator, and the validation, or otherwise, of a state change proposal. If a

proposed change is invalidated, the proposer's coordinator will rollback their local object state using a call to applyState with the previously agreed state. A similar process to that outlined applies to update, as opposed to overwrite, of object state.

In this case, the *B2BObjectController* update operation is used to indicate the type of state coordination required. The examine operation indicates that object state will be read but not written in the current scope. Together with enter and leave, the three access type indication operations (examine, overwrite and update) can be used as hooks for concurrency control mechanisms and transactional access to objects.

The implementation of the *B2BObjectController* is provided as part of the middleware. Together, *B2BObject* and *B2BObjectController* provide connection management; state change scoping and access type indication; and upcalls for application-level validation. connect and disconnect operations initiate connection to and disconnection from the set of objects being coordinated (leading to initiation of connection and disconnection protocols via the *B2BCoordinatorLocal* propagation interface). validateConnect and validateDisconnect allow application-specific validation of connection and disconnection requests.

The semantics of connect, disconnect and leave vary with the communication mode. In synchronous mode, they block until the relevant coordination process completes (an exception is raised if validation fails). In asynchronous mode, they return immediately and completion is signalled by the coordinator through invocation of coordCallback. In deferred synchronous mode they return immediately and a blocking call to coordCommit can be used to wait for completion. coordCallback is also used by the coordinator to communicate protocol progress information to the application.

The *B2BCoordinatorLocal* interface is independent of both the communication mode and the coordination protocols executed between coordinators through their *B2BCoordinatorRemote* interface. Implementations of these interfaces are part of the *B2BCoordinator* package provided by the middleware.

6.3 X-Contract Implementation with B2BObjects

With this background, we can begin with hinting at the overall implementation of x-contracts using the B2BObjects middleware. The implementation of an x-contract that involves a purchaser and a supplier is shown in figure 6.3. Each party maintains a copy of the contract object, encoded as one or more B2BObjects (B2Bobj); operations on these objects are controlled by the contract FSMs. The dashed line that goes from the supplier to the purchaser shows what happens when the supplier sends an offer. When the offer is ready, the supplier invokes a send operation, and the supplier's FSM switches to its *Waiting for response* state

and makes a *SendOffer* call to the local copy of a shared B2Bobj (that implements the operation). The local B2Bobj collects, and signs, evidence of the operation and requests coordination of the proposed update to its state with the purchaser's B2Bobj.

The purchaser's B2Bobj verifies the evidence provided and makes an up-call to the purchaser's FSM to validate the B2Bobj operation. Upon receiving the up-call, the purchaser's FSM switches to the *Deciding to buy state*.

The dashed line from the purchaser's FSM to the supplier's FSM shows how the purchaser's response is transmitted to the supplier. The B2BObjects middleware ensures that all operations performed by the purchaser and the supplier are recorded and are non-repudiable. Thanks to this facility the purchaser of the example of Fig 6.3, can provide evidence, at a court for example, that he sent his payment within 7 days after receiving a notification of acceptance of his purchase order, even if the supplier denies receiving the payment. One of the major advantages of B2BObjects is that it ensures this without the need of involving centralized trusted third parties.

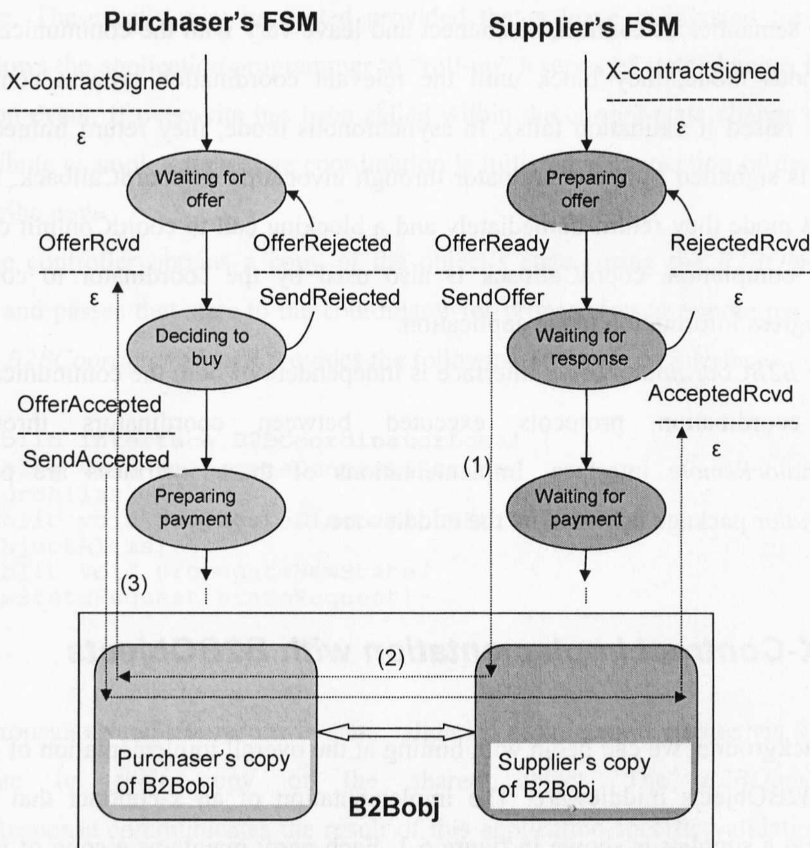


Fig.6.3. Collection of non-repudiable digital evidence with B2Bobjects

Contract management must be made part of the business processes of the organizations involved. An organization's business processes can be divided into two broad categories. (a)

The business processes that are internal to the organization, and (b) the contract management processes, that involve interactions with trading partners.

In our contract model enterprises that engage in contractual relationships are autonomous and wish to remain autonomous after signing a contract. Thus a signing enterprise has its own resources and local policies. In our view each contracting enterprise is a black box where private business processes represented as finite state machines, workflows or similar automaton, run.

A private business process interacts with its external environment through the contract from time to time to influence the course of the shared business process.

Thus, a contract is a mechanism that is conceptually located in the middle of the interacting enterprises to intercept all the contractual operations that the parties try to perform. Intercepted operations are accepted or rejected in accordance with the contract clauses.

6.4 Purchaser/Supplier Example

In this section, we continue with the implementation of the Purchaser/Supplier contract example that we presented in Chapters 3, and 4.

To remind the reader, we first began in Chapter 3, with presenting a contract between a Purchaser and a Supplier in which the Supplier is entitled to make price offers for a certain commodity he wishes to sell. The Purchaser in turn is entitled to reject or accept these offers.

1 Offer
1.1 The supplier may use his discretion to send offers to the purchaser.
1.2 If no offer is sent within seven days after the signature of the x-contract, or after the latest rejected offer, the x-contract shall be terminated.
1.3 The purchaser is entitled to accept or reject the offer, but he shall notify his decision to the supplier within five days after the receipt of the offer.
2 Commencement and completion
2.1 The contract shall start immediately upon signature.
2.2 The purchaser and the supplier shall terminate the x-contract immediately after reaching a deal for buying an item.

Fig. 6.4. Contract clauses after removal of ambiguities

From the contract, we extracted the rights and obligations of each of the contract signatories, and mapped them into FSMs. In Chapter 4, we used the Spin model checker to validate the correctness of the contract FSMs with respect to a number of safety and liveness contract correctness requirements. During the validation process, inconsistencies were found within

the contract. These were corrected, and the essential contract clauses are now as presented in figure 6.4. Also the corrected FSMs are in figure 6.5.

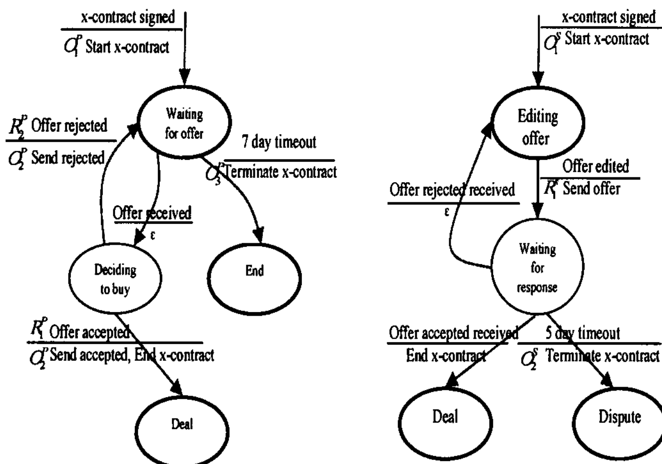


Fig.6.5. Corrected Purchaser and Supplier FMSs

Our next and final task is to implement the contract in figure 6.4, and its resulting finite state machines in figure 6.5, using the B2BObjects middleware service, and thus creating the x-contract.

As we mentioned in Section 3.4.3, the optimum contract implementation scenario would be one in which the signatories to a contract, and/or their lawyers, can convert a contract into an x-contract that monitors and enforces the agreement without requiring the expertise of a technical person. This however is not possible with current technology, and each newly agreed contract will require the involvement of computing personnel with experience in the area of x-contract validation and implementation.

This ideal scenario however can be achieved in the business world of standard contracts (Section 3.4.3). X-contract computing experts will initially be involved in the process of validating and creating standard x-contracts to implement the original standard contracts. Once the standard x-contracts have been created, they can be stored together with their standard contracts on a web space to be accessed by interested clients, and they will not require the involvement of technical persons because the inconsistency free x-contract would have already been created.

For our example, we will take the view that the contract above is required for use as a standard contract. The main text of the contract will remain the same for various clients who wish to use the contract, however specific details (such as: The date the contract will be implemented, and the maximum allowable number of days until the supplier makes an offer, etc.) will be determined as required by the clients signing the contract.

6.4.1 Implementation of Supplier/Purchaser Example

Our objective is to convert the FSMs in Fig 6.5, to a Java representation of the FSMs that can be interpreted an executed as an x-contract by the B2BObjects middleware.

B2BObjects implementation relies on operations performed on the shared information encapsulated within local object replicas. Within the context of x-contracts, these operations are aimed to progress the state of the object replicas in consistency with the requirements of the signed contract. This progress or update of the state of an object by one of the parties is subject to non-repudiable validation by each party. We must first begin with identifying the object(s) within the contract in Fig 2.4 on which we will require to perform the operations that will update the state of the x-contract. It is clear from the FSMs in Fig 6.5, that all operations are related to the offer being made by the supplier. And remembering the B2BObjects API description in Section 6.2, we can proceed with creating a B2BOffer object. We have chosen to extend an existing application object (AbstractB2BObject) that implements the B2BObject interface. We present the B2BOffer Class next:

```
1    import java.util.Date;
2
3    import uk.ac.ncl.b2bobj.AbstractB2BObject;
4    import uk.ac.ncl.b2bobj.B2BException;
5    import uk.ac.ncl.b2bobj.B2BInvalidatedException;
6    import uk.ac.ncl.b2bobj.B2BObjectController;
7
8    public class B2BOffer extends AbstractB2BObject {
9
10       public B2BOffer(Date contractDate, B2BObjectController ctrlr)
11           throws B2BException {
12           offer = new Offer(contractDate);
13           setController(ctrlr);
14       }
15
16       public void setPrice(double price) throws B2BException,
17           B2BInvalidatedException {
18           ctrlr.enter();
19           ctrlr.overwrite();
20           offer.setPrice(price);
21           ctrlr.leave();
22       }
23
24       public void accept() throws B2BException,
B2BInvalidatedException {
25           ctrlr.enter();
26           ctrlr.overwrite();
27           offer.accept();
28           ctrlr.leave();
29       }
30
31       public void reject() throws B2BException,
B2BInvalidatedException {
32           ctrlr.enter();
33           ctrlr.overwrite();
```

```
34          offer.reject();
35          ctrlr.leave();
36      }
37
38
39      public Date getContractDate() throws B2BException,
40          B2BInvalidatedException {
41          ctrlr.enter();
42          ctrlr.examine();
43          Date cdate = offer.getContractDate();
44          ctrlr.leave();
45
46          return cdate;
47      }
48
49      public Date getOfferDate() throws B2BException,
B2BInvalidatedException {
50          ctrlr.enter();
51          ctrlr.examine();
52          Date odate = offer.getOfferDate();
53          ctrlr.leave();
54
55          return odate;
56      }
57
58      public double getPrice() throws B2BException,
B2BInvalidatedException {
59          ctrlr.enter();
60          ctrlr.examine();
61          double price = offer.getPrice();
62          ctrlr.leave();
63
64          return price;
65      }
66
67      public Date getResponseDate() throws B2BException,
68          B2BInvalidatedException {
69          ctrlr.enter();
70          ctrlr.examine();
71          Date rdate = offer.getResponseDate();
72          ctrlr.leave();
73
74          return rdate;
75      }
76
77      public boolean isAccepted() throws B2BException,
B2BInvalidatedException {
78          ctrlr.enter();
79          ctrlr.examine();
80          boolean accept = offer.isAccepted();
81          ctrlr.leave();
82
83          return accept;
84      }
85
86      /**
87       * @see B2BObject#applyState
88       */
89      public void applyState(Object state) throws B2BException {
90          this.offer = (Offer) state;
91      }
```

```
92
93       /**
94        * @see B2BObject#getState
95        */
96       public Object getState() throws B2BException {
97           return offer;
98       }
99
100      private Offer offer;
101 }
```

The various operations on the B2BOffer object are encapsulated within this Class, for example accepting and rejecting an offer (lines 24, 31), or getting the offered price (58) by the Purchaser, or the Supplier making an offer in the setPrice method (line 16). Also this Class contains methods to apply a new state to the object, and to get the latest state of the object (87, 96).

Once we have determined the details of the OfferObject, all that remains is to convert the FSM diagrams in Fig 6.5, into Java code that will perform the operations in OfferObject at the right time. We will suffice here with showing the constructors and the runContract methods for both the purchaser and the supplier Classes that contain the FSMs. The FSM for the Purchaser begins in line 64, and for the Supplier begins in line 65.

Purchaser Class

```
50       public Purchaser(Date contractDate, PurchaserStateMachine
                                                             psm,
51           boolean verbose) throws Exception {
52           this.psm = psm;
53           this.verbose = verbose;
54
55           B2BObjectController ctrlr = new B2BObjectControllerImpl(
56                   ./etc/purchaser.properties");
57
58           offer = new B2BOffer(contractDate, ctrlr);
59
60           ctrlr.addB2BValidationListener((B2BValidationListener)
                                                                psm);
61           ctrlr.addB2BEventListener((B2BEventListener) this);
62       }
63
64       public void runContract() throws Exception {
65           boolean tryAgain = true;
66
67           while (tryAgain) {
68               psm.waitForOffer();
69
70               printContract();
71
72               tryAgain = false;
73
74               if (psm.getState() ==
```

```
                    PurchaserStateMachine.DECIDING_TO_BUY) {
75                  BufferedReader console = new BufferedReader(
76
                            new InputStreamReader(System.in));
77                  System.out.print("Offer price is: "
78                                        + offer.getPrice()
79                                        + " accept it (y/n)? ");
80
81                  String ans = console.readLine();
82
83                  try {
84                      if (ans.toLowerCase().equals("y")) {
85                          offer.accept();
86                          psm.setState(PurchaserStateMachine.DEAL);
87                      } else {
88                          offer.reject();
89                          psm.setState(
90
                    PurchaserStateMachine.WAITING_FOR_OFFER);
91
92                          tryAgain = true;
93                      }
94                  } catch (B2BInvalidatedException e) {
95                      psm.setState(PurchaserStateMachine.END);
96                  } catch (Exception e) {
97                      e.printStackTrace();
98                      throw e;
99                  }
100             }
101
102             printContract();
103         }
104
105         System.out.println("contract terminating ...");
106         offer.getController().disconnect();
107         System.exit(0);
108
109     }
```

Supplier Class

```
51      public Supplier(Date contractDate, SupplierStateMachine ssm,
52          boolean verbose) throws Exception {
53          this.ssm = ssm;
54          this.verbose = verbose;
55
56          B2BObjectController ctrlr = new B2BObjectControllerImpl(
57                              ./etc/supplier.properties");
58
59          offer = new B2BOffer(contractDate, ctrlr);
60
61          ctrlr.addB2BValidationListener((B2BValidationListener)
                                                           ssm);
62          ctrlr.addB2BEventListener((B2BEventListener) this);
63      }
64
65      public void runContract() throws Exception {
66          waitForMemberJoin();
```

```
67
68          while (ssm.getState() ==
                        SupplierStateMachine.EDITING_OFFER) {
69              printContract();
70              BufferedReader console = new BufferedReader(
71              new InputStreamReader(System.in));
72              System.out.print("Enter offer price: ");
73              String price = console.readLine();
74
75              ssm.setResponseDeadline();
76
77              try {
78                  offer.setPrice(Double.parseDouble(price));
79                  ssm.waitForResponse(true);
80              } catch (B2BInvalidatedException e) {
81                  ssm.setState(SupplierStateMachine.DISPUTE);
82              } catch (Exception e) {
83                  e.printStackTrace();
84                  throw e;
85              }
86
87              printContract();
88          }
89
90          System.out.println("contract terminating ...");
91          offer.getController().disconnect();
92          System.exit(0);
93
94      }
95
```

Once the three main classes (B2BOffer, Purchaser, and Supplier) and any other support Classes have been compiled, we can proceed with the implementation.

As we stated earlier, we are assuming that our contract is intended for use as a standard contract therefore we have included some additional Classes that create a simple interface which can be used to tailor certain contract data to the requirements of the contract signatories. The interface can be seen in Fig 6.6.

Fig 6.6, Simple Example of a Contract Editor

While the structure of an x-contract cannot be safely modified without the knowledge of lawyers and technical experts, at least the contract parties can set the data that makes up the x-contract without requiring the intervention of these experts. An x-contract editor such as the one in Fig 6.6, can be used to store and access numerous contracts with their ready made x-contracts. Once the data has been entered, the contract can be saved, and then loaded and implemented at the agreed time by the signatories. See Fig 6.7.

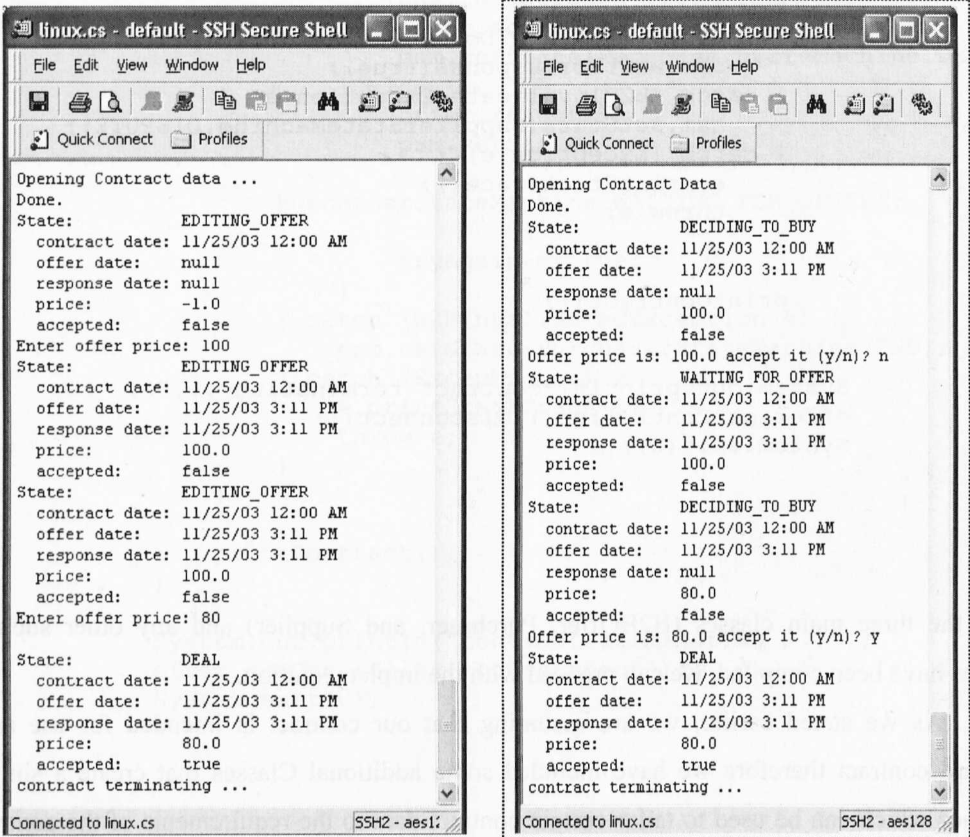


Fig 6.7 Sample implementation of an x-contract

Figure 6.7, shows a possible sequence of events during implementation of the x-contract in Fig 6.4. The states through which the parties pass are clearly labelled as are the dates at which these states occurred. In this implementation, the parties begin with loading the contract data that was saved when editing the contract data (Fig 6.6). The Supplier now begins at the EDITING-OFFER state and sends the purchaser a price offer of 100. At the DECIDING_TO_BUY state, the purchaser rejects the supplier's initial offer and his response is sent to the Supplier. The Supplier modifies his offer which is subsequently accepted by the Purchaser, and they both go to the DEAL state.

All of these transactions between the two parties are attempts at updating the data within OfferObject of which each of the signatories has an identical copy. An update however

does not occur unless all parties agree to that update, or the update does not contradict the terms of the contract. This example shows how in some cases the opposing party may veto the B2BObject update (for example the Purchaser disagrees with an offer being made), and in other cases the x-contract itself will refuse an object update, for example if the purchaser responds after an agreed time limit has passed.

6.5 Summary

We have presented middleware that addresses the requirement for dependable information sharing between organisations. The middleware presents the abstraction of shared state and regulates updates to that state. Safety is guaranteed even in the presence of misbehaving parties. If all parties behave correctly, liveness is guaranteed despite a bounded number of temporary failures. The middleware presents a familiar programming abstraction to the application programmer and frees them to concentrate on the business logic of applications. B2BObjects is used to regulate the interaction between the contracting parties and to collect non-repudiable evidence of each of their actions. Using B2BObjects, x-contracts can be monitored and enforced without requiring the involvement of independent trusted third parties.

A question that we do not cover and which requires further research is the issue of access control. A contract management system and its resources must be protected from unauthorised access, disclosure, modification or destruction of its services and its information. This can only be accomplished by ensuring, among other things, that the identification and authentication of the legitimate users of system services and their encapsulated resources are securely verified. Therefore a topic for further research is the development of a contract management system that implements Role-based Access Control (RBAC) architecture [FSG01]. The basic concept of RBAC is that entities (users, machines, services, etc.) in each enterprise of a VE (Virtual Environment) are assigned to roles, permissions are assigned to roles, and entities acquire permissions by being members of roles. An RBAC model architecture such as the one developed by OASIS [BMY01] [YMB02] for example, could be a promising approach to achieve the requirements of security and trust within the context of access control.

Chapter Seven

Summary and Future Work

This thesis has proposed an approach for electronically executable contract (x-contract) representation, validation, implementation, and monitoring. It has employed the use of finite state machines for the modelling process, and thus benefiting from the employment of the widely used Spin model validation tool, for the validation of the correctness and consistency of contracts.

For the x-contract verification process, this thesis proposes a list of contract correctness requirements, and for the contract implementation phase, we suggest the use of the novel B2BObjects middleware service that provides for the requirements of safety, liveness, and non-repudiation.

7.1 Contract Modelling with Finite State Machines (Chapter 3)

Before attempting to implement an x-contract electronically; the clauses within the original conventional text contract must be precisely abstracted and the parties' rights and obligation must be mapped into computer code convertible mathematical notation, also the ambiguities that exist within the original conventional text contract must be detected and removed.

To specify party interaction related rights and obligations, we propose the use of finite state machines. Thanks to their graphic nature, finite state machines are easy to read. On the other hand the mathematical theory behind them makes them useful for ensuring the correct operation of an x-contract.

In this chapter we described and proposed a method by which contracts' rights and obligations can be mapped into FSMs.

7.2 Validation of electronic contracts (Chapter 4)

It is crucial that we identify and eliminate the ambiguities that exist within the clauses of a text contract before it can be implemented electronically.

In this chapter, we have introduced the protocol modelling language Promela, and the protocol verification tool Spin. We have analysed with the aid of simple examples the correctness properties that must be satisfied for a contract to be correct.

Based on our analysis we have developed a list of correctness requirements that we suggest that x-contract designers use during the contract validation process:

CR1: Correct commencement: An x-contract should start its execution in a well-defined initial state on a specific date or when something happens.

CR2: **Correct termination:** An x-contract should reach a well-defined termination state on a specific date or when something happens.

CR3: **Reachability**: Each and every state within an x-contract should be reachable, i.e. executable at least in one of the execution paths of the x-contract.

CR4: **Freedom from deadlocks:** An x-contract should never enter a situation in which no further progress is possible.

CR5: **Partial correctness:** If an x-contract begins its execution with a precondition true then, the x-contract will never terminate with the precondition false, regardless of the path followed by the x-contract from the initial to its final state

CR6: **Invariant:** If an x-contract begins its execution with a precondition true then, the precondition should remain true for the whole duration of the contract.

CR7: **Occurrence or accessibility:** A given activity should be performed by an x-contract at least once no matter what execution path the x-contract performs.

CR8: **Precedence:** An x-contract can perform a certain activity only if a given condition is satisfied.

CR9: **Absence of livelocks:** The execution of an x-contract should not loop infinitely through a sequence of steps that has been identified as undesirable, presumably because the sequence produces undesirable output or no output at all.

CR10: **Responsiveness:** The request for a service will be answered before a finite amount of time.

CR11: **Absence of unsolicited responses**: An x-contract should not allow a contractual party to send unsolicited responses.

These correctness requirements are split into safety and liveness properties, and we summarize them as:

- Safety properties: reachability, partial correctness, invariant, deadlocks, precedence, absence of unsolicited responses.
- Liveness properties: correct termination, occurrence, livelocks, responsiveness.

We further categorize Safety properties into, *general* safety properties that must hold true for any x-contract (CR3: Reachability, CR4: Freedom from deadlocks, CR11: Absence of unsolicited responses), and *specific* safety properties that must hold true only if so required by

the contracting parties for the specific requirements of a certain x-contract (CR5: Partial correctness, CR6: Invariant, and CR8: Precedence).

Contracting parties may desire complex correctness requirements that are a combination of a number of the above requirements. Fortunately, it has been formally proven that any correctness property can be represented as the intersection of a safety property and a liveness property [AS85].

7.3 Modelling and Verifying the Correctness of Contracts; Examples (Chapter 5)

We present three different examples of text based documents (contracts) containing rules that govern the interaction between different parties. Through these examples, we demonstrate the ideas of contract representation with finite state machines, and contract validation with Spin, developed in Chapter 3, and Chapter 4.

In this chapter, we convert the text based contracts into executable contract models through a process that removes the ambiguities that may exist in the original text contracts. This is to facilitate the correct implementation of the x-contracts at run time.

There are many examples, where the interaction between two or more parties, over a network, calls for a set of rules that can be implemented to police this interaction.

In cases where the rules of interaction need to be negotiated and agreed upon by the interacting parties, the rules constitute contract clauses, which will combine to form a contract that the parties must sign. This is the bases for the first and second examples.

There are cases however where the interaction between the parties is governed by rules that are already in place. The parties need only to understand them and agree upon them before the interaction can begin. Our third example reflects this case. We present the scenario where two or more parties are involved in a game that is played over the Internet.

In our three examples, we carefully use the safety and liveness contract correctness requirements proposed in Chapter 4 to detect and remove ambiguities that are present within the clauses of the original contracts. General safety properties are properties that must be checked for in any contract for it to be free from ambiguities. Specific safety properties and liveness properties are properties that are checked for within a contract only if so required by the contracting parties.

After ambiguities are detected and removed from a contract model, it can be converted into program code for implementation.

7.4 Middleware Support for X-Contract Implementation (Chapter 6)

In this chapter, we present the B2BObjects middleware service that addresses the requirement for dependable information sharing between organisations. The middleware presents the abstraction of shared state and regulates updates to that state.

Safety is guaranteed even in the presence of misbehaving parties. If all parties behave correctly, liveness is guaranteed despite a bounded number of temporary failures. The middleware presents a familiar programming abstraction to the application programmer and frees them to concentrate on the business logic of applications. B2BObjects is used to regulate the interaction between the contracting parties and to collect non-repudiable evidence of each of their actions. Using B2BObjects, x-contracts can be monitored and enforced without requiring the involvement of independent trusted third parties.

7.5 For Future Work

We believe that the principles and techniques developed in this thesis represent logical steps which when applied correctly will lead to the execution of free from ambiguity correct(ed) executable contracts that are capable of monitoring the interactions stipulated within the clauses of the contract.

Future work for the time being will concentrate on the application and testing of the ideas within this thesis at an industry level.

Our work is currently based on the implementation of agreed and fixed contracts, and takes no account of the possibility that the clients may wish to make changes to its content and therefore to the content of the x-contract during execution time. This work therefore needs to be progressed further as it is not yet clear how our contract management model can be deployed dynamically and made to respond to changes.

Another area needing further work (mentioned in Section 6.5) is integration of advanced RBAC techniques, such as developed in OASIS [BMY01] that provide extended notions of appointments, for delegation of role-playing, and of multiple, mutually aware domains for mobile roles, that can be re-located and still able to communicate without confusion.

The availability of a contract management service creates a safe and secure way for organisations to form Virtual Organisations (VOs) that provide new composite services (CSs). Clearly there needs to be a common standard between organisations for specifying, publishing, finding and composing CSs. Indeed, emerging Web services standards such as SOAP, UDDI, WSDL etc, as well as grid related work on Open grid Services (OGSA) are steps in this

direction. Unfortunately, they do not yet fully address issues of services that can be made available, scalable and adaptive.

Some of these issues are being addressed by several industry led efforts at developing standards for specifying, composing and coordinating the execution of CSs. These include ebXML/OASIS [EBXML], Web services architecture work at W3C [WSA], and Rosettanet [RIF00]. In any case, certain basic facilities for contract representation and monitoring will be required. The work presented here provides a sound foundation for future developments.

Blank Page

References

[AEB01] Abrahams A.S., Eyers D.M., and Bacon J.M. *Mechanical Consistency Analysis for Business Contracts and Policies.* Proc 5th International Conference on Electronic Commerce Research (ICECR5), Montreal, Canada, 23-27 October 2002. Society 2001.

[AG01] Angelov S, Grefen P. *B2B eContract Handling-A Survey of Projects, papers, and Standards.* University of Twene, The Netherlands. 2001.

[AS85] Alpern B, Schneider F.B, *Defining liveness, Information Processing Letters,*Vol. 21, N. 4, Oct, 1985.

[AS03] *A Survey of Legal Issues Relating to the Security of Electronic Information.* http://canada.justice.gc.ca/en/ps/ec/summary.html. Department of Justice, Canada. 2003.

[BIZ] http://www.biztalk.org

[BMY01] Bacon J, Moody K, and Yao W, *Access Control and Trust in the use of Widely Distributed Services,* IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), November 2001, Heidelberg,, Lecture Notes in Computer Science. VOL. 2218, pp. 300-315.

[BP98] Baum M S, Perritt H H Jr. *Electronic contracting, publishing, and EDI law.* Wiley Law Publication: 6-7. 1998.

[BPML] http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[CF00] *CrossFlow Project.* http://www.crossflow.org/. 2000.

[CL00] Chomicki J, Lobo J. *A Logic Programming Approach to Conflict Resolution in Policy Management.* Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000), Breckenridge, Colorado, USA, Morgan Kaufmann. 2000.

[CO99] *COSMOS Project.* http://vsys-www.informatik.uni-hamburg.de/projects/cosmos/index.phtml. 1999.

[CSP99] COSMOS Project. http://vsys-www.informatik.uni-hamburg.de/projects/cosmos/index.phtml. 1999.

[CSW02] Cook N, Shrivastava S.K, and Wheater S.M, *Distributed Object Middleware to Support Dependable Information Sharing between Organisations,* Proc. IEEE Int. Conf. on Dependable Systems. and Networks (DSN-2002), Bethesda USA, June 2002.

[D98] Daoud F, *A Business Contracting Model for TINA Architecture, Electronic Markets,* International Journal of Electronic Markets, Vol.8 No.3 1998, University of St.Gallen, Switzerland.

[D00] Daskalopulu A. *Modelling Legal Contracts as Processes. Legal Information Systems Applications*, 11th International Conference and Workshop on Database and Expert Systems Applications, IEEE C. S. Press, pp. 1074–1079, 2000.

[DBSL02] Damianou N, Bandara A.K, Sloman M, and Lupu E.C. *A Survey of Policy Specification Approaches*. Department of Computing Imperial College. April 2002.

[DD01] Damianou N, Dulay N.*The Ponder Policy Specification Language. Policy 2001:* Workshop on Policies for Distributed Systems and Networks, Bristol, UK, Springer-Verlag. 2001.

[DDLS01] Damianou N, Dulay N, Lupu E, Sloman M. *The Ponder Policy Specification Language.* In Proc. Int. Workshop on Policies for Distributed Systems and Networks (POLICY), Bristol, UK, Springer-Verlag LNCS 1995, Jan. 2001.

[DDM01] Daskalopulu A, Dimitrakos T, and Maibaum T, *E-Contract Fulfilment and Agents' Attitudes*. Proceedings of ERCIM WG E-Commerce Workshop on The Role of Trust in e Business, Zurich, October, 2001.

[DDNS98] Dan A, Dias D, Nguyen T, Sachs M, Shaikh H, King R, Duri S. *The Coyote Project: Framework for Multi-party E-Commerce*, Proceedings of the 7th Delos Workshop on Electronic Commerce, Crete, Greece, Sept. 21-23, 1998.

[ECO] http://eco.commerce.net

[EDI03] *What is EDI?* http://www.x12.org/x12org/about/index.html?whatis.html. 2003.

[EBXML] http://www.ebxml.org

[EER] http://www.cs.sfu.ca/CC/354/zaiane/material/notes/Chapter2/node1.html

[FHK95] Fox D, Horster P, Kraaibeek P. *Grundüberlegungen zu Trust Centern.* In: Horster, P. (Ed.): Trust Center; DuD; 1995.

[FKNT02] Foster I, Kesselman C, Nick J, Tuecke S, *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration.* Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[FKT01] Foster, I., Kesselman, C. and Tuecke, S. *The Anatomy of the Grid: Enabling Scalable Virtual Organizations.* International Journal of High Performance Computing Applications, 15 (3). 200-222. 2001. www.globus.org/research/papers/anatomy.pdf.

[FSG01] Ferraiolo D.F, Sandhu R, Gavrila S, Kuhn D.R, and Chandramouli R, *Proposed NIST standard for Role-Based Access Control*, ACM transactions on Information and System Security, Vol. 4, No. 3, Aug. 2001.

[GBW98] Griffel F, Boger M, Weinreich H, Lamersdorf W, Merz M, *Electronic Contracting with COSMOS - How to Establish, Negotiate and Execute*

Electronic Contracts on the Internet, Proceedings from 2nd Int. Enterprise Distributed Object Computing Workshop (EDOC '98), 1998.

[GGKS02] Gotschalk K, Graham S, Kreger H, and Snell J. *Introduction to Web Services Architecture*. 2002. IBM Software Group. IBM Systems Journal, vol. 41, No. 2, pp. 170-177, 2002.

[GJS99] Gisler M, Johri Y, Schopp B. *Requirements on Secure Electronic Contracts*. University of St. Gallen Switzerland. 1999.

[GM00] Goodchild A, Herring C, and Milosevic Z. *Business Contracts for B2B*. Proceedings of the CAISE*00 Workshop on Infrastructure for Dynamic Business-to-Business Service Outsourcing, Stockholm, June 5-6,2000.

[GRID] www.grid.org

[GSG00] Gisler M, Stanoevska-Slabeva K, Greunz M, *Legal Aspects of Electronic Contracts*, Infrastructures for Dynamic Business-to-Business Service Outsourcing (IDSO'00), Stockholm, June 5-6, 2000.

[GSS00] Greunz M, Schopp B, Stanoevska-Slabeva K, *Supporting Market Transactions through XML Contracting Container*, Proceedings of the Sixth Americas Conference on Information Systems (AMCISS 2000), Long Beach, CA USA, 10-13 August, 2000.

[H91] Holzmann G.J. *Design and Validation of Computer Protocols*. Prentice Hall, 512 pgs. 1991.

[H97] Holzmann G.J. *Basic Spin Manual*. http://spinroot.com/spin/Man/Manual.html. 1997.

[HLGG00] Hoffner Y, Ludwig H, Gülcü C, Grefen P. *Architecture for Cross-OrganisationalBusiness Processes,* Proceedings 2nd International Workshop on Advanced Issues of Ecommerce and Web-Based Information Systems, Milpitas, CA, USA, 2000, pp. 2-11.

[IBMW] http://www.ibm.com/software/solutions/webservices/resources.html

[ISO99] ISO/IEC. Information Technology - Open Distributed Processing Reference Model - Enterprise Viewpoint. 1999.

[JS97] Jajodia S, Samarati P. *A Logical Language for Expressing Authorisations*. IEEE Symposium on Security and Privacy, Oakland, USA, IEEE. 1997.

[KBCS00] Keen P, Balance C, Chan S, Schrump S, *Electronic Commerce Relationships: Trust by Design*, Prentice Hall PTR, 2000.

[KGV00] Koetsier M, Grefen P, Vonk J. *Contracts for Cross-Organizational Workflow Management*, Proceedings 1st International Conference on Electronic Commerce and Web Technologies, London, UK, 2000, pp. 110-121.

[L77] Lamport L, *Proving the correctness of multiprocess programs*.

March 1977. IEEE Transactions on Software Engineering, SE-3(2).

[L98] Lee R.M, *Towards Open Electronic Contracting, Electronic Markets*, International Journal of Electronic Markets, Vol.8 No.3 1998, University of St. Gallen, Switzerland.

[L00] LEE M. *Event and Rule Services for Achieving a Web-based Knowledge Network*. Computer and Information Science and Engineering, University of Florida. 2000.

[LFK] http://www.legal-forms-kit.com/

[LN99] Lobo J R, Naqvi B S. *A Policy Description Language*. 1999. *AAAI*, Orlando, Florida.

[LTL] http://www.time-rover.com/ftp/tl.pdf

[LR97] Lindemann M.A, Runge A. *Non-Repudiation within the Electronic Contracting Phase of Electronic Commerce Transactions*. Conference Proceedings of the First Overcoming Barriers to Electronic Commerce Conference OBEC'97, Malaga, Spain, April 1997.

[LR98] Lindemann M, Runge A. *Electronic Contracting within the Reference Model for Electronic Markets*, Proceedings of the 6th European Conference on Information Systems ECIS '98, Aix-en-Provence, France, June 4-6, 1998.

[LRP97] Lindemann M.A, Runge A, *Permanent IT-Support in Electronic Commerce Transactions, Electronic Market Architectures*, International Journal of Electronic Markets ,Vol. 7, No. 1, 1997.

[LS97] Lpupu, E. C, and Sloman M.S. *Towards a Role Based Framework for Distributed Systems Management*. Journal of Network and Systems Management **5**(1): 5-30. 1997.

[LS98] Lindemann M, Schmid B.F. *Elements of a Reference Model for Electronic Markets*, Proceedings of the 31st Annual Hawaii International Conference on Systems Science HICCS'98, Vol. IV, pp. 193-201, Hawaii, January 6-9, 1998.

[LS99] Lupu E. C, and Sloman M.S. *Conflicts in Policy-Based Distributed Systems Management*. In IEEE Transactions on Software Engineering - Special Issue on Inconsistency Management 25(6): 852-869. 1999.

[LTSA99] Labelled Transition System Analyser. http://www.doc.ic.ac.uk/~jnm/book/ltsa/LTSA.html. 1999.

[MAO96] Milosevic Z, Arnold D, O'Connor L. *Inter-enterprise Contract Architecture for Open Distributed Systems: Security 'Requirements*. WET ICE'96 Workshop on Enterprise Security, Stanford, USA, June 1996.

[MB95] Milosevic Z, Bond A. *Electronic Commerce on the Internet: What is Still Missing?* Proc. 5th Conf. of the Internet Society, pp.245-254, Honolulu, June 1995.

[MD02] Milosevic Z, Dromey R G. *On Expressing and Monitoring Behaviour in Contracts*. In proceedings of the 6th International Enterprise Distributed Object Computing Conference (EDOC2000), Lausanne, Switzerland, Sep. 17-20, 2002.

[MM01] Marjanovic O, and Milosevic Z. Towards Formal Modelling of e-Contracts. Proceedings of the Fifth IEEE International Enterprise Distributed Object Computing Conference, Seattle, Washington, September 04-07, 2001.

[MMU01] Minsky N, Minsky Y, and Ungureanu V, *Safe Tuplespace-Based Coordination in Multi Agent Systems*, in the Journal of Applied Artificial Intelligence (AAI), January 2001 (Vol 15, No. 1, pages: 11-33).

[MQ] MQSeries workflow. http://www-3.ibm.com/software/integration/wmqwf/.

[MSS03*] Molina-Jimenez C, Shrivastava S, Solaiman E, and Warne J. *Run-time Monitoring and Enforcement of Electronic Contracts*. Submitted to ECRA (Electronic Commerce Research and Applications) for publication.

[MSSW03*] Molina-Jimenez C, Shrivastava S, Solaiman E, and Warne J. *Contract Representation for Run-time Monitoring and Enforcement*. Proc. IEEE Int. Conf. on E-Commerce (CEC-2003), Newport Beach, California, June 2003.

[MU00] Minsky N, Ungureanu V. *Law-Governed Interaction: A Coordination & Control Mechanism for Heterogeneous Distributed Systems*. ACM Transactions on Software Engineering and Methodology (TOSEM), (Vol 9, No 3, pages: 273-305). July 2000.

[MU01] Minsky N.H, Ungureanu V. *Scalable Regulation of Inter-enterprise Electronic Commerce*, In Proceedings of the Second Ineternational Workshop, WELCOM 2001 Heidelberg, Germany, November 2001. Lecture Notes in Computer Science, Vol. 2232, Springer.

[NC00] Naumovich G, Clarke L.A, *Classifying Properties: An Alternative to the Safety-Liveness Classification*, In Proceedings of the Eighth International Symposium on the Fundations of Software Engineering, Nov. 2000.

[OASIS] http://www.oasis-open.org

[OEDI] *Open EDI*. http://www.euridis.nl/weboutline/Web.OpenEDI.html. Erasmus University, Netherlands.

[P02] Ponder: A Policy Language for Distributed Systems Management http://www-dse.doc.ic.ac.uk/Research/policies/ponder.shtml. 2002.

[PROT] http://www.w3.org/Protocols/

[R98] Ranno F. *A Language and Toolkit for the Specification, Execution and Monitoring of Dependable Distributed Applications*. PhD Thesis. The University of Newcastle upon Tyne. 1998.

[RA98] Runge A, *The Need for Supporting Electronic Commerce with Electronic Contracting*, Proceedings of the Conference on Information Systems and Technology (INFORMS), Montreal, Canada, April 1998.

[RL98] Ronald M. Lee. *Towards Open Electronic Contracting, Electronic Markets.* International Journal of Electronic Markets, Vol.8 No.3 1998, University of St. Gallen, Switzerland.

[RM-ODP] http://www.dstc.edu.au/Research/Projects/ODP/ref_model.html

[RSKS99] Runge A, Schopp B, Stanoevska-Slabeva K, *The Management of Business Transactions through Electronic Contracts*, Proceedings for the of the 10th International Workshop on Database and Expert Systems Applications (DEXA'99), Florence, September 1999.

[S01] Sookman B. *Computer, Internet and Electronic Commerce Law,* Carswell Thomson Publishing, 2001: 10-1.

[S02] Sakharuk D. *History of Electronic Contracting.* http://www.kentlaw.edu/classes/rwarner/legalaspects_ukraine/contracting/commentary/history/history_electronic_contracting.htm. 2002.

[SCIO03] SearchCIO.com Definitions http://searchcio.techtarget.com/sDefinition/0,,sid19_gci213925,00.html. 2003.

[SD00] Steen M W A, Derrick J. *ODP Enterprise Viewpoint Specification. Computer Standards and Interfaces* 22: 65-189. 2000.

[SL01] Stanley Y W, LAM H, Minsoo L, Bai S, Shen Z. *An Information Infrastructure and E-services for Supporting Internet-based Scalable E-business Enterprises.* 5th IEEE Annual Enterprise Distributed Object Conference (EDOC2001), Seattle, WA, IEEE Computer Society. 2001.

[SOAP] http://www.w3.org/TR/SOAP/

[SMS*] Solaiman E, Molina-Jimenez C, Shrivastava S, *Model Checking Correctness Properties of Electronic Contracts.* International Conference on Service Oriented Computing (ICSOC03), Trento, Italy, December 2003.

[SP03] Spin. http://spinroot.com/spin/whatispin.html. 2003.

[SXM01] Serban C, Xuhui A, and Minsky N, *Establishing Enterprise Communities,* In Proc. of the 5th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2001), Seattle Washington, September 2001.

[TINA02] Telecommunications Information Networking Architecture consortium. http://www.tinac.com. 2002.

[UM00] Ungureanu V, and Minsky N, *Establishing Business Rules for Inter-Enterprise Electronic Commerce*, In Proc. of the 14th International Symposium on DIStributed Computing (DISC 2000), LNCS, No. 1914, pages 179-193, Springer-Verlag, October 2000, Toledo Spain

[UNCE] http://www.uncefact.org

[WSDL] http://www.w3.org/TR/wsdl

[XMNU00] Xuhui A, Minsky N, Nguyen T, Ungureanu V, *Law-Governed Communities Over the Internet.* In Proc. of Coordination' 2000: Fourth International Conference on Coordination Models and Languages, LNCS, No. 1906, pages 133-147, Springer-Verlag, September 2000, Limassol Cyprus.

[YAW98] Soon-Yong Choi, Dale O. Stahl, and Andrew B. Whinston, *Intermediation, Contracts and Micropayments in Electronic Commerce*, Electronic Markets, International Journal of Electronic Markets, Vol.8 No.1 1998, University of St. Gallen, Switzerlan.

[YMB02] Yao W, Moody K, and Bacon J, *A Model of OASIS Role-Based Access Control and its Support for Active Security*, ACM Trans. On Information and System Security, 5, 4, November 2002.

Note: References with the star [*] sign, are own publications.